## Contents

1 Library structure 3

2 Reporting errors and information 4

3 Data 6
   3.1 Data header 6
   3.2 Data representation 7
   3.3 Representation and statistics of data tables 8
   3.4 Data formats 9
      3.4.1 CSV 9
      3.4.2 ARFF 10
      3.4.3 RSES2 11
   3.5 Loading and saving data 11
   3.6 Representation of value distributions and vectors 12

4 Computational methods 13
   4.1 Setting configuration parameters 13
   4.2 Calculation statistics 15
   4.3 Reporting progress and measuring time 15
   4.4 Constructing and passing data structures 16

5 Classifiers and testing 18
   5.1 Defining classifiers 18
   5.2 Visualization 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Defining rule classifiers</td>
<td>19</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Rules</td>
<td>19</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Generating rules</td>
<td>21</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Defining rule classifiers</td>
<td>21</td>
</tr>
<tr>
<td>5.4</td>
<td>Testing classifiers and classification results</td>
<td>22</td>
</tr>
<tr>
<td>5.5</td>
<td>Simultaneous tests of many classifiers</td>
<td>22</td>
</tr>
<tr>
<td>5.6</td>
<td>Crossvalidation and iterative tests</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Classifier types</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Rough set based rule classifier</td>
<td>25</td>
</tr>
<tr>
<td>6.2</td>
<td>$K$ nearest neighbours / RIONA</td>
<td>28</td>
</tr>
<tr>
<td>6.3</td>
<td>$K$ nearest neighbours with local metric induction</td>
<td>30</td>
</tr>
<tr>
<td>6.4</td>
<td>Decision tree C4.5</td>
<td>31</td>
</tr>
<tr>
<td>6.5</td>
<td>Rule classifier AQ15</td>
<td>32</td>
</tr>
<tr>
<td>6.6</td>
<td>Neural network</td>
<td>33</td>
</tr>
<tr>
<td>6.7</td>
<td>Naive Bayes</td>
<td>34</td>
</tr>
<tr>
<td>6.8</td>
<td>Support vector machine</td>
<td>35</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Description:</td>
<td>35</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Parameters</td>
<td>35</td>
</tr>
<tr>
<td>6.8.2.1</td>
<td>Kernel transformations</td>
<td>36</td>
</tr>
<tr>
<td>6.8.3</td>
<td>Hints</td>
<td>37</td>
</tr>
<tr>
<td>6.9</td>
<td>Classifier based on principal components analysis</td>
<td>37</td>
</tr>
<tr>
<td>6.10</td>
<td>Classifier based on local principal components analysis</td>
<td>38</td>
</tr>
<tr>
<td>6.11</td>
<td>Bagging</td>
<td>38</td>
</tr>
<tr>
<td>6.12</td>
<td>AdaBoost</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>Tools</td>
<td>40</td>
</tr>
<tr>
<td>7.1</td>
<td>Weka</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>Graphical user interface</td>
<td>41</td>
</tr>
<tr>
<td>7.3</td>
<td>Computing in cluster</td>
<td>41</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Experiment definition</td>
<td>42</td>
</tr>
<tr>
<td>7.4</td>
<td>Command line programs</td>
<td>43</td>
</tr>
</tbody>
</table>

Bibliography 44
Chapter 1

Library structure

Rseslib library consists of the 5 main packages: rseslib.system, rseslib.util, rseslib.structure, rseslib.processing, rseslib.example. Besides the rsestool package contains various tool for the library: graphical interface and a program for distributing computations in a network.

rseslib.system package  The most basic package providing classes and methods for communication with an underlying operating system. It manages library configuration, and errors, progress and computation results reporting.

rseslib.util package  The package providing useful methods extending standard Java library.

rseslib.structure package  The first one from the two major library packages. It covers all the structures representing data and their models, as well as mathematical objects and functions, constructed and used while running rseslib computations.
Each structure should provide saving and reading from a file by implementation of java.io.Serializable interface.
Each structure should also provide text representation by implementation of the standard method toString().

rseslib.processing package  The second one from the two major library packages. It covers all data processing methods. Most of them transforms one data representation into another one.

rseslib.example package  The package providing examples of the rseslib library usage.
Chapter 2

Reporting errors and information

The most basic library class is the Report class (the rseslib.system package). It directs messages of various types to the appropriate places. In rseslib there are 3 kinds of messages.

Error messages The Report class handles errors appearing while library procedures are running. If the procedure catches an exception (e.g. DataFormatException that means bad data format), it can transfer it up or handle and report the error occurrence by calling the function Report.exception(Exception) and providing the caught exception as an argument.

The function exception reports errors to the connected outputs. The function Report.addErrorOutput(Output) connects outputs for error reporting. With the help of this function one can connect any number of outputs e.g. StandardErrorOutput writing information to the standard error output, FileOutput writing messages to a given file (both classes are provided in the rseslib.system.output package) or any other outputs defined by a library user.

Information messages Information messages enable to communicate about the effects and the results of a running program. Like error messages they can be written to the console, to a file or to a graphical interface window defined by a user. The method Report.addInfoOutput(Output) connects outputs for information messages. One can write messages by calling Report.display(Object) and Report.displaynl(Object). These functions use the standard method toString() what enables to write information about any structure provided in the library. One can turn information messages off and on in a whole program or in a particular part of a program by calling Report.setInfoDisplay(false) and Report.setInfoDisplay(true) respectively.
Debugging messages  Debugging messages are used only for library objects development and they are used to provide control messages that verify the correctness of implemented algorithms. Like for error and information messages any number of outputs can be connected with the use of the function `Report.addDebugOutput(Output)`. The function `Report.debugn1(String)` writes debugging messages whereas the function `Report.setDebugDisplay(boolean)` turns debugging messages on and off in the selected parts of a program.

Warning! If any channel for error, debugging or information messages was connected, at the end of a program it is good to use `Report.close()` to close all the used channels.
Chapter 3

Data

Data are set of objects that are described by vectors of attribute values. A set of attributes for a given set of objects is fixed. Data header describing the attribute types and the data objects are the main components of data representation.

3.1 Data header

All classes related to header representation are provided in the rseslib.structure.attribute package.

A data header represents a set of attributes describing data. The Header interface provides the basic header functionality. The function noOfAttr() returns the number of attributes. The attributes in a header are indexed from 0 to noOfAttr()-1. The function attribute(int i) in the Header interface returns representation of the i-th attribute that implements the Attribute interface.

Each attribute has its name name(). Attributes are categorized according to the data type: each attribute is either numerical (isNumeric() returns true and then the attribute is represented by an object of the NumericAttribute class) or symbolic (isNominal() returns true and then the attribute is represented by an object of the NominalAttribute class). Attributes are categorized also according to the meaning for classification: each attribute is either conditional (isConditional() returns true) or decision (isDecision() returns true) or it does not matter for classification ((isText() returns true).

The header representation accepts zero or many decision attributes in data. In the case of the often situation that one attribute is selected to be a symbolic decision and the rest are conditional attributes there are the two additional functions in the Header interface providing the index of the decision attribute (the decision() method) and the representation of this attribute (the nominalDecisionAttribute() method).
All data structures related to a given header implement the `Headerable` interface (the `resel.lib.structure` package) and they keep reference to the same header object implementing the `Header` interface.

### 3.2 Data representation

The classes for single data objects representation are provided in the `resel.lib.structure.data` package.

The `DoubleDataWithDecision` interface that extends the `DoubleData` interface is the basic interface for representation of an object with a single decision in the form of an attribute value vector. Attribute values in this interface are represented by the `double` type. `set(int attr, double val)` and `get(int attr)` are the functions for setting and getting attribute values, `setDecision(double val)` are `getDecision()` are the functions for setting and getting the decision value. Each data object is related to its header which is provided by the function `attributes()`.

Numerical attribute values (`attributes().isNumeric(attr)==true`) are represented immediately by the `double` type values, e.g.

```java
DoubleData dData;
;
System.out.println('Attr. 0 value =' + dData.get(0));
```

Symbolic values (`attributes().isNominal(attr)==true`) are read first to the `String` type and then they are encoded by the successive integer numbers from 0 up. The dictionary, which transformes value codes to the corresponding texts and back, is available with the help by the objects of the `NominalAttribute` class, e.g.

```java
DoubleData dData;
String value;
;
NominalAttribute attr0 = (NominalAttribute)dData.attributes().attribute(0);
dData.set(0, attr0.globalValueCode(value));
System.out.println('Attr. 0 value =' + NominalAttribute.stringValue(dData.get(0));
```

Warning! The codes of symbolic values are unique for the whole system, e.g. there is one dictionary encoding symbolic values global for the whole system. It means that the codes of the same symbolic value are the same even if objects have different headers.
Because of efficiency many computational methods interprets symbolic values as indices in an array, e.g. while computing distribution of attribute values in data. To enable this the NominalAttribute structure provides local value coding from 0 to noOfValues()-1 where noOfValues() denotes the number of values occurring in data on this attribute. The functions localValueCode(double globalValueCode) and globalValueCode(int localValueCode) can be used to change a global value code to local and vice versa.

Missing attribute values, occurring in the situations when provided data are incomplete or attribute values in some object are undefined, are represented as the values Double.NaN both for numerical and symbolic attributes. To check whether an attribute value is undefined and to compare it with the attribute values of other objects the function Double.isNaN(double attrVal) must be used. The operators ==, !=, < <= can be used only to defined attribute values.

Warning! You cannot serialize global values codes. Global coding is connected to system instance. There are two possibilities:

- you can save original strings representing symbolic values; then you can get codes during objects re-reading,
- you can save local values codes; serialization of header and symbolic attributes saves local value codes.

Attributes have different meanings during classification: conditional ((attributes().isConditional(attr)==true)), decision ((attributes().isDecision(attr)==true)) and it does not matter for classification ((attributes().isText(attr)==true)). Using values of attributes in data objects you need to remember that conditional, decision and meaningless attributes are provided in the same way - for example, if you need only conditional attributes, you need to check it: ((attributes().isConditional(attr)==true)), decision ((attributes().isDecision(attr)==true)) and meaningless ((attributes().isText(attr)==true)).

```java
DoubleData dDat;
;
for (int attr = 0; attr < dDat.attributes().noOfAttr(); attr++)
    if (dDat.attributes().isConditional(attr)) {
        // use value of attr attribute
    }
```

### 3.3 Representation and statistics of data tables

Classes for representing data tables are provided in `rseglib.stucture.table` package.
The basic interface for representing a set of data objects is `DoubleDataTable`. The interface provides methods: adding to the set (`add(DoubleData dData)`), removing an object from the set (`remove(DoubleData dData)`) and checking the number of objects in the set (`noOfObjects()`). Standard object data processing:

```
DoubleDataTable table;
::
    for (DoubleData dData : table.getDataObjects()) {
        ::
    }
```

If you have data with only one symbolic decision, the interface `DoubleDataTable` assumes that the class realizing it computes decisions distribution in the objects set. The number of decisions and local value codes are included in the structure which represents decision attribute. This structure is returned by the method `attributes().nominalDecisionAttribute()` and decision distribution for local value codes is returned by the method `getDecisionDistributionForLocalCodes()`.

You can also get general information about table with decision distribution in a text form by the method `toString()`.

The table provides also basic statistics about numeric values (minimum, maximum, mean and standard deviation of attribute in the table). Statistics are provided by the method `getNumericalStatistics(int)`.

Warning! The standard implementation of interface `DoubleDataTable` is class `ArrayListDoubleDataTable`.

### 3.4 Data formats

Rseslib reads 3 data formats: CSV, ARFF and RSES2.

#### 3.4.1 CSV

Comma Separated Version - popular format of spreadsheet programs like Excel, OpenOffice Calc, etc.

Values may be separated by comma and/or whitespaces. Comments (lines starting with `'#'`) and empty lines are allowed.

This format requires additional header describing data. A header may be provided in two ways:
• by adding the header at the beginning of a data file, in this case the header must be enclosed with the lines indicating the beginning and the the end of the header. The line starting the header must contain the `beginheader` tag, the line ending the header must contain the `endheader` tag.

• by providing a separate file with the header; this option eliminates the inconvenience of editing large files in case of very large data sets.

Header file contains description of all attributes, one attribute per one header line. Optionally, it can contain a separate line defining missing value string. Comments (lines starting with ‘#’) and empty lines are also allowed.

Each attribute line starts with an attribute name. Next two keywords are required. The first keyword defines the role of the attribute: conditional is used by classifiers in learning, decision is the attribute to be guessed by classifiers, text is only a descriptive attribute, doesn’t take part in learning and classification. The abbreviations c, d, t may be used instead of the full role name. The second keyword defines the type of the attribute values: numeric or nominal. The abbreviations nu, no may be used instead of the full type name. Optionally one can add skip as the third keyword - then the attribute is ignored and not loaded.

Missing values are defined in the optional line starting with the keyword `missing_value` and followed by the list of strings (separated by comma and/or whitespaces) representing missing value. It means it is possible to have more than one notation for missing value in single data file.

Example of a CSV header:

```
# my comment

missing_value    ?

id                text, nominal, skip
attr1             conditional, numeric
attr2             conditional, nominal
:
class             decision, nominal
```

Exemplary data sets in CSV format along with headers are available at http://rsproject.mimuw.edu.pl.

### 3.4.2 ARFF

Format of the very popular open source machine learning library WEKA. The format description is available in WEKA documentation at http://www.cs.waikato.ac.nz/ml/weka. To load an arff file Weka jar must be provided in the class path while starting your program which uses Rseslib, e.g.:
java -cp ...weka.jar...

You can take Weka jar from Weka installation directory.

3.4.3 RSES2

Format of RSES2 system. The format description is available in RSES2 documenta-

3.5 Loading and saving data

Data loading is realized by one of ArrayListDoubleDataTable class construc-
tors.

To load data from the file containing the data header (CSV including a header, 
ARFF and RSES2) use the constructor ArrayListDoubleDataTable(File,Progress), 
e.g.

    DoubleDataTable table = new ArrayListDoubleDataTable(
        new File('data/heart.dat'),
        new StdOutProgress());

The second argument is a progress reporting object.

To load data from CSV file(-s) with a separate header file load first the header 
using the constructor ArrayHeader(File), e.g.

    Header hdr = new ArrayHeader(new File('data/segment.hdr'));

and then use 3-argument constructor ArrayListDoubleDataTable(File,Header,Progress), 
e.g.

    DoubleDataTable table = new ArrayListDoubleDataTable(
        new File('data/segment.trn'),
        hdr,
        new StdOutProgress());

Warning. The same header object may be used to load multpile CSV files of 
the same type, e.g.
Header hdr = new ArrayHeader(new File("data/segment.hdr"));
;
DoubleDataTable trnTable = new ArrayListDoubleDataTable(
    new File("data/segment.trn"),
    hdr,
    new StdOutProgress());
DoubleDataTable tstTable = new ArrayListDoubleDataTable(
    new File("data/segment.tst"),
    hdr,
    new StdOutProgress());

It is possible to load only a header from files containing both the header and data by using the same constructor `ArrayHeader(File)` as for header files.

Data can be saved in two formats: CSV and ARFF. Such saved data can be loaded again using one of `ArrayListDoubleDataTable` constructors.

To save data in CSV format use the method `store(File,Progress)`. The CSV format is saved with the rseslib header added at the beginning of file.

To save data in ARFF format use the method `storeArff(String,File,Progress)`. The first argument is the name of a data set which is required by ARFF format.

3.6 Representation of value distributions and vectors

`Vector` class is used to represent distributions of attribute values, especially decisions distributions, and to represent vectors in a vector space (`rseslib.stucture.vector`). This class integrates features of values distribution (it provides functions, which are necessary for counting values) and vectors distribution (it provides a large set of operations on vectors).
Chapter 4

Computational methods

4.1 Setting configuration parameters

Classes which implement complex computational methods, can provide configuration parameters of represented methods, for example selection of metric and number of neighbours for k-nearest neighbours method or selection of measure for decision trees cutting and tree depth. Parameters for configurable methods are represented in rseslib by java.util.Properties class objects. The class which can configure parameters should satisfy two conditions:

- Provide configuration file with default parameters values; the file should be located in the same package as the configurable class and should have the same name as this class with the file extension .properties, for example for class rseslib.processing.classification.knn.KNNClassifier the correct path for configuration file is rseslib/processing/classification/knn/KNNClassifier.properties

The successive lines contain either parameter values definition or comments (lines beginning with # character):

```
# metric's type
metric = CityWithSVD
# number of neighbours
k = 3
```

- Inherit from class Configuration (package rseslib.system). The first parameter for class constructor is dictionary with parameters values. This dictionary should be passed when calling Configuration constructor, for example:

```java
Configuration conf = new Configuration();
conf.setParameters(new Properties()
```

```java
    .setProperty("metric", "CityWithSVD")
    .setProperty("k", "3")
```

```java
    .build();
```

```java`
public class KNNClassifier extends Configuration {
    public KNNClassifier(Properties prop, ...) {
        super(prop);
        ...
    }
    ...
}

The parameters can be passed to configurable class object in two ways:

- by defining parameters in configuration file and creating object with default parameters values: if there is null as a first parameter in object constructor, the object loads default parameters values from configuration file, e.g.

    knn = new KNNClassifier(null, ...);

- by defining parameters in program and passing them to the constructor of configurable class object, e.g.

    Properties knnProp = Configuration.loadDefaultProperties(
        KNNClassifier.class);
    knnProp.setProperty('metric', 'SVD');
    knnProp.setProperty('k', '5');
    knn = new KNNClassifier(knnProp, ...);

The first solution makes permanent change of configuration parameters values. All successive created objects of this class have new parameter values. The second solutions does not change default values in configuration file.

One can get parameters value by calling `getProperty(String propertyName)` function in an object of configuration class. In case of the boolean, integer or real parameters, the value of parameter one can get by calling one of thee following functions: `getBoolProperty`, `getIntProperty` or `getDoubleProperty`.

Warning! To avoid forcing implementation of interface `java.io.Serializable` in all inheritable classes of class `Configuration`, the class `Configuration` does not implement this interface, but provides serialization methods: `writeConfiguration(ObjectOutputStream)` and `readConfiguration(ObjectInputStream)`. Implementing serialization in a class, which is an inheritance of `Configuration` class, you have to redefine default implementation of serialization and call this two methods.
4.2 Calculation statistics

Some parameters are calculated during complex calculations execution, e.g. number of rules or number of nodes in the tree. The class `ConfigurationWithStatistics` (package `rreslib.system`) allows such statistics viewing. The class that extends class `ConfigurationWithStatistics` can calculate statistics and save them by calling (for all calculated values) the function `addToStatistics(String name, String value)` in the body of the method `calculateStatistics()` for every computable value. Statistics set can be available (e.g. for objects, which can compute statistics) by calling function `getStatistics()`.

Warning! To avoid forcing implementation of interface `java.io.Serializable` in all inheritable classes of class `ConfigurationWithStatistics`, class `ConfigurationWithStatistics` does not implement this interface, but provides serialization methods: `writeConfigurationAndStatistics(ObjectOutputStream)` and `readConfigurationAndStatistics(ObjectInputStream)`. Implementing serialization in a class, which is an inheritance of `ConfigurationWithStatistics` class, you have to redefine default implementation of serialization and call this two methods.

4.3 Reporting progress and measuring time

Computational methods, which can execute at least several seconds, should report calculations progress. All classes to reporting calculations progress are in package `rreslib.system.progress`.

The basic interface for reporting is `Progress`. The constructor or method realizing complex calculation should get as a parameter the object implementing interface `Progress`. The interface has two functions: the first `set(String name, int noOfSteps)` sets calculations name and number of steps. You should call it once - before calculations beginning. The second function `step()` is helpful for reporting successive steps execution, e.g.

```java
public KNNClassifier(..., Progress prog) {
    prog.set('Indexing of the set', <<train set size>>);
    for every train objects
        index it
        prog.step();
}
```

The standard interface implementation is class `StdOutProgress`. Informations about progress in this class are written to standard output - they show percen-
tage calculations completion. It is written if calculations increase at least by 2%. Typical using this class:

```java
tkn = new KNNClassifier(., new StdOutProgress());
```

If progress reporting is not necessary, the object of class `EmptyProgress` can be passed as a parameter. Then calling functions `set` and `step` gives no results.

Sometimes you want progress reporting from several calculations as one summary process. In this case you can use object `MultiProgress`. It is "an agent" between multiple calculations and other progress reporting object, e.g. object of class `StdOutProgress`. During constructing object of class `MultiProgress` you pass destination progress reporting object and table, which defines number of steps of every calculation. In the following example there is defined object `MultiProgress` for 3 calculations; it transformes them to the single 100-steps calculation:

```java
int[] computations = new int[3];
comutations[0] = 10;
computation[1] = 30;
computations[2] = 60;
Progress prog = new MultiProgress('Sum',
    new StdOutProgress(), computations);
```

It is available to measure calculations time during computational methods testing. You need to use class `Timers` in package `rserb.use.time`. The class provides as many timers as you want: they can start, stop, restart and reset measuring time. The class provides both time: from timer running and from the last restart.

### 4.4 Constructing and passing data structures

Complex computational methods, e.g. classifiers and discretizers often use complex data structures, e.g. k-nearest neighbours method uses metric definition and a tree that indexes objects set. There is assumed that all subsidiary structures are passed as parameters to the constructor of class, which implements computational methods. It is possible that the class has a few constructors - it depends on programmer who chooses subsidiary structures preparation level while he is using implementation of computational method, for example one can define two constructors for k-nearest neighbours method: one of them for users, who provide metric definition; second one for users, who want the method to induce the metric from training data:
public KNNClassifier(Properties prop, Metric metr,
                        DoubleDataTable trainingSet) {
    // the method uses metric passed by the user
    // to index train set
    :
}
public KNNClassifier(Properties prop,
                        DoubleDataTable trainingSet) {
    // the method induces metric from train set
    // and use it to index this set
    :
}
Chapter 5

Classifiers and testing

The interfaces and abstract classes for defining and testing classifiers are included in the rseslib.processing.classification package.

5.1 Defining classifiers

The basic interface for classifiers which assign unique decision for object data is Classifier. It has 3 functions: classify(DoubleData dObj) for classification of single object, calculateStatistics() to make statistics and resetStatistics() to reset statistics. Typical classifier implements this interface and extends abstract class ConfigurationWithStatistics:

```java
public KNNClassifier extends ConfigurationWithStatistics
    implements Classifier {
    :
}
```

Data structures are created by constructor of class, which implements this constructor. Every classifier should provide 3-parameter constructor, which create classification model on the basis of train table:

```java
public KNNClassifier(Properties, DoubleDataTable, Progress) {
    :
}
```
To make saving and loading classifier from file, it has to implement interface \texttt{java.io.Serializable}.

Sometimes user can provide some classifier’s components and then he has to define his add-in constructors (see 4.4).

Then he has to define three functions. The function \texttt{classify(DoubleData)} returns generated by classifier decision for object as global value code (default all objects have global coded decision, see 3.2). Function \texttt{calculateStatistics()} is used to saving counted statistics and usually include calling of function \texttt{addToStatistics} for every counted value (see 4.2). Function \texttt{resetStatistics()} restarts statistics counting.

Classifier with generalized decision is kind of classifier which assigns distribution of decision (instead of a single decision) to objects. To define classifier with distributed decision use \texttt{ClassifierWithDistributedDecision} interface, which include \texttt{classifyWithDistributedDecision(DoubleData)} method instead of \texttt{classify(DoubleData)} method. The \texttt{classifyWithDistributedDecision(DoubleData)} method returns as a result the vector of decisions weights.

Warning! During implementation of functions \texttt{classify, classifyWithDistributedDecision}, and using local values code for symbolic attributes (see 3.2) remember, that classified object can have symbolic attribute values, which do not occur in training data set, i.e. in the set used to construct classifier.

\section*{5.2 Visualization}

Every classifier can implement graphical presentation. To see such classifier you can use Quak system (see 7.2).

Classifier with grafical presentation implements interface \texttt{VisualClassifier}. It is available in \texttt{rselib.processing.classification} package. Besides methods providing by \texttt{Classifier} interface, it has two add-in methods. The method \texttt{draw(JPanel canvas)} draws classifier on graphical panel. Method \texttt{drawClassify(JPanel canvas, DoubleData obj)} presents classification of single object (graphical classification presentation can be different from classifier presentation).

\section*{5.3 Defining rule classifiers}

\subsection*{5.3.1 Rules}

The rules are defined in \texttt{rselib.structure.rule} package. There are two basic interface to implementation of rules defined: \texttt{PartialMatchingRule} and
Rule. Both interfaces are defined for deterministic rules, it means both interfaces have one fixed decision `getDecision()`. Also both interfaces have the function, which score object to rule matching. Interface `PartialMatchingRule` is more general: matching function can have any values from \([0; 1]\). The interface `Rule` is more specific and is used for rules, which have only two possibilities: either object matches the rule or not.

For users, who want to create typical rules, which allow only two possibilities of matching object, the class `BooleanFunctionRule` is implemented. To create a rule, in constructor of class `BooleanFunctionRule` you shoul pass two parameters: boolean function `BooleanFunction` deciding if object match to the rule or not; and rule decision.

`BooleanFunction` is an interface defined in `rseslib.structure.function.booleanval` package. This package includes implementation of the most typical boolean functions used for rules construction:

- **AttributeEquality**: defines single descriptor, which compares fixed attribute to fixed value
- **AttributeInterval**: defines single descriptor for numeric attributes, which checks if value belongs to fixed values interval
- **AttributeValueSubset**: defines single descriptor, which checks if value belongs to fixed values subset
- **Conjunction, Disjunction**: defines conjuction and disjunction of boolean functions, like descriptors
- **Negation**: defines negation of boolean function

To create rules you can use these function and also - create your own functions, implemented for more atypical rules. Typical rule definition is:

```
DoubleDataWithDecision obj;

booleanFunction[] descr = new booleanFunctions[3];
descr[0] = new AttributeEquality(1, obj.get(1));
descr[1] = new AttributeEquality(4, obj.get(4));
descr[1] = new AttributeEquality(7, obj.get(7));
Rule r = new BooleanFunctionRule(new Conjunction(descr),
    obj.getDecision());
```

Rules, that have two parameters: accuracy and support, which are inducted from training set, should additionaly implement interface `RuleWithStatistics`.

Each rule generated from Rseslib provides text representation, available by the method `toString()`.
5.3.2 Generating rules

The methods generating rules from table or other data structure are contained in \texttt{rreslib.processing.rules} package. The basic interface for rules generating methods is \texttt{RuleGenerator}. The implementation of rules generating methods demands function \texttt{generate(DoubleDataTable tab)} defining, which returns generated rules set.

The exemplary implementation of rules generator is class \texttt{AccurateRuleGenerator}. It generates a set of accurate rules for respective objects; it means that for every object the class generates one, maximal specific rule, matching to this object.

For generators based on different data structure (e.g. reduct, decision tree), you should define another interface for such kind of generator.

5.3.3 Defining rule classifiers

You can define rule classifiers in library in \texttt{rreslib.processing.classification.rules} package. The basic classifiers defined in one file, can be installed directly in the package. For more complex classifiers you should create subpackage. Defining rule classifiers is the same as in case of other classifiers (see 5).

Typical rule classifier either uses existing rules generator, passed as a parameter to the constructor, e.g.

```java
public RuleBasedClassifier extends ConfigurationWithStatistics implements Classifier {
    // Constructor...
    public RuleBasedClassifier(Properties prop,
                              RuleGenerator gen, DoubleDataTable trnTab) {
        // Implementation...
    }
}
```

or creates new generator inside the constructor. The classifier uses projection on the type \texttt{Rule, RuleWithStatistics} or another - it depends on kinds of used rules.

The exemple of rule classifier is \texttt{MajorityClassifierWithRules}. The classifier uses accurate rules’ generator \texttt{AccurateRuleGenerator} (see 5.3.2) and during object classification - if there exists a matching rule, it returns decision of the matching rule; if not, it returns majority decision from training set.
5.4 Testing classifiers and classification results

The class `SingleClassifierTest` is used for testing of table. Objects of the class provide function `classify(Classifier cl, DoubleDataTable testTable, Progress prog)`. The classification progress is reported by using object `prog`. The function returns classification results for table `testTable` as a `TestResult` object. Function `getAccuracy()` in this object returns classification’s accuracy for all objects, and function `getDecAccuracy(double dec)` returns classification’s accuracy for objects with chosen decision `dec`. Accuracy is fraction of correctly classified objects (real value from $[0, 1]$). Global accuracy, decision accuracy and calculation statistics are available as a text by `toString()` method. Furthermore the method `getNoOfObject(double realDec, double assignedDec)` for every decision value returns information how many objects with decision `realDec` were classified to decision `assignedDec`.

5.5 Simultaneous tests of many classifiers

The library allows simultaneously testing of many classifiers. One can use `ClassifierSet` object, which represents any defined by user classifiers set. Every classifier can be added to the set by `addClassifier(String name, Class classifierType, Properties prop)` method, passing any name as a parameter. Additionally it has two functions for classifiers testing: `train(DoubleDataTable, Progress)`, which is used for classifiers creating and training, and `classify(DoubleDataTable, Progress)`, which is used for testing of created classifiers. Exemplary usage of the object:

```java
DoubleDataTable trainTable, testTable;

ClassifierSet mySet = new ClassifierSet();
mySet.addClassifier('Default Decision Tree', C45.class, null);
mySet.addClassifier('Default KNN', KNNClassifier.class, null);
mySet.train(trainTable, new StdOutProgress());
Map<String, TestResults> results =
    mySet.classify(testTable, new StdOutProgress());
```

The classification result is a dictionary, where key is a name of classifier (passing as a parameter of a function `addClassifier(...)`) and value is the classification result for this classifier, represents by `TestResult` object (see 5.4). All results (whole dictionary) you can for example write to log file using function from the class `Report`:
Both functions `train(...)` and `classify(...)` can be used many times. When you use the function `train(...)` again, the classifiers are reconstructed on new table.

The object `ClassifierSet` uses default classifiers constructor. To add a classifier (created by any constructor) to the set, use the method `addClassifier(String name, Classifier cl)`, which adds trained classifier. Classifiers, which were added by this method, are skipped in training method `train(...)`, but they are testing as other classifiers by the method `classify(...).

### 5.6 Crossvalidation and iterative tests

The library provides also tools for more complex tests: cross-validation (`CrossValidationTest`), iterative tests with random partition of table at each test (`MultipleRandomSplitTest`) and iterative cross-validation (`MultipleCrossValidationTest`). Typical using of both tools is very simple: you create the object by passing the set of classifiers and call testing function:

```java
DoubleDataTable testTab;
ClassifierSet mySet;
:
Properties prop = Configuration.loadDefaultProperties(
    CrossValidationTest.class);
    prop.setProperty('noOfFolds', '10');
CrossValidationTest cvt = new CrossValidationTest(prop, mySet);
Map<String, MultipleTestResult> results =
    cvt.test(testTab, new StdOutProgress());
```

In case of the two other kinds of tests, the objects representing tests are handled in the same way. The classification results are a dictionary, where key is a name of classifier and value is the classification result for this classifier, represented by `MultipleTestResult` object. The object provides average accuracy from all test and standard deviation of accuracy. Both parameters are available as a text by using `toString()` method. The results (whole dictionary) you can for example write to log file using function from the class `Report`:

```java
Report.displayMapWithMultiLines('Tests results', results)
```
All kinds of tests have testing parameters. **CrossValidationTest** has one parameter: the number of folds. **MultipleRandomSplitTest** have three parameters: the first is number of tests, the second and the third define a factor: size of training table divided by size of test table during random partition of original table. **MultipleCrossValidationTest** has two parameters: the number of tests and the number of cross-validation folds at each test.
Chapter 6

Classifier types

6.1 Rough set based rule classifier

Authors: Rafał Latkowski, Marcin Jachmuzna, Michał Kurzydlowski, Krzysztof Niemkiewicz, Dariusz Ogórek, Marcin Piliszcuk, Beata Zielosko

Rseslib class path:
rseslib.processing.classification.rules.roughset.VisualRoughSetClassifier

Weka class path:
weka.classifiers.rules.RoughSet

Description:
The classifier is based on rough sets [14, 22]. It uses the concepts of discernibility matrix, reducts and rules generated from reducts [22]. It enables to apply a discretization method to numeric attributes from a range of available discretization types. It provides modes to work with incomplete data (with missing values) [10, 24], with inconsistent data [15, 21] and implements more effective local reducts [29] for larger data sets.

The classifier is available in Weka after installing Rseslib package (see Section 7.1). One-letter options of the classifier displayed in Weka reports are explained after running (on Linux use colon, on Windows use semicolon to separate jar paths):

```
java -cp [path-to-weka.jar]:[path-to-rseslib.jar]
   weka.classifiers.rules.RoughSet -h
```

One can see visualization of the classifier using Qmak system (see Section 7.2). Visualization of the classifier presents all the generated rules. Visualization
of single object classification presents the rules that match the object to be classified.

**Parameters:**

- **Discretization** - defines type of discretization applied to numerical attributes. Description and experimental comparison of available discretization methods can be found in [7]:
  - *None* - the classifier does not use discretization
  - *EqualWidth* - the range of values of a numerical attribute is divided into intervals of equal length
  - *EqualFrequency* - the range of values of a numerical attribute is divided into intervals containing the same number of training objects
  - *OneRule* - Holte’s 1R algorithm [6], tries to cut the range of values into interval containing training objects with the same decision but avoids very small intervals
  - *EntropyMinimizationStatic* - top-down method, at each step searching for a cut with the greatest information gain (the greatest entropy reduction) [4], the cuts are generated separately for each numerical attribute
  - *EntropyMinimizationDynamic* - top-down method, at each step searching for a cut with the greatest information gain (the greatest entropy reduction) [4], the best cut is searched over all attributes
  - *ChiMerge* - bottom-up method, merging intervals when they have similar values of statistical measure $\chi^2$ describing the proportion of decision classes in an interval [9]
  - *MaximalDiscernibilityHeuristicGlobal* - top-down method, at each step searching for a cut that discerns the greatest number of pairs of training objects with different decisions [25], the global method searches for a cut maximizing discernibility in all subset of training objects at once
  - *MaximalDiscernibilityHeuristicLocal* - top-down method, at each step searching for a cut that discerns the greatest number of pairs of training objects with different decisions [25], the local method searches for a cut maximizing discernibility in a single subset of training objects

- **DiscrNumberOfIntervals** - discretization parametr used only if *Discretization* is set to * EqualWidth* or *EqualFrequency*. It defines the number of intervals for each numerical attribute.

- **DiscrMinimalFrequency** - discretization parametr used only if *Discretization* is set to *OneRule*. It defines the minimal number of training objects that must fall into each interval generated by 1R algorithm.
DiscrConfidenceLevelForIntervalDifference $(0.0 - 1.0)$ - discretization parameter used only if Discretization is set to ChiMerge. It defines the confidence level required to consider two neighbouring intervals as different and not to merge them by ChiMerge method.

DiscrMinimalNumberOfIntervals - discretization parameter used only if Discretization is set to ChiMerge. It defines the minimal number of intervals for each numerical attribute generated by ChiMerge method.

Reducts - method generating reducts from discernibility matrix. Global methods generate reducts and rules for all training objects together. Local methods generate reducts and rules for each training object separately.

- AllLocal - generates all local reducts [29], uses an efficient algorithm generating prime implicants from boolean CNF formula [2, 20]
- AllGlobal - generates all global reducts [22], uses an efficient algorithm generating prime implicants from boolean CNF formula [2, 20]
- OneJohnson - generates one reduct by greedy Johnson’s algorithm [8, 13]
- AllJohnson - generates all reducts that may be obtained from Johnson’s algorithm [8, 13]
- PartialLocal - generates partial local reducts [12]
- PartialGlobal - generates partial global reducts [12]

IndiscernibilityForMissing - defines how missing values are treated when classifier compares objects [10]:

- DiscernFromValue - missing value is different from all other values; indiscernibility relation is symmetric and transitive
- DontDiscernFromValue - missing value is indiscernible from other values; indiscernibility relation is symmetric, but not transitive
- DiscernFromValueOneWay - asymmetric discernibility; missing value is indiscernible from other values, but values are different from missing value; indiscernibility relation is asymmetric and intransitive

DiscernibilityMethod - defines which pairs of objects are discernible during generation of discernibility matrix [22]:

- OrdinaryAndInconsistenciesOmitted - discernibility matrix is constructed for pairs of object with different decisions, but the pairs which have different decisions but are indiscernible on other attributes are omitted.
- GeneralizedDecision - discernibility matrix is constructed for pairs of object with different generalized decisions [15, 21]. It eliminates problems with discernibility of objects with different original decisions, but indiscernible on other attributes (such objects have the same generalized decisions).
– *GeneralizedDecisionAndOrdinaryChecked* - discernibility matrix is constructed only for pairs, which have different both decisions: generalized and original.

– *All* - discernibility matrix is constructed for all pairs except for indiscernible objects.

*GeneralizedDecisionTransitiveClosure* (TRUE/FALSE) - is used only if *DiscernibilityMethod* is set to *GeneralizedDecision* or *GeneralizedDecisionAndOrdinaryChecked*. If missing value is indiscernible from other values (*IndiscernibilityForMissing≠DiscernFromValue*) indiscernibility relation on data with missing values is not an equivalence relation and ”ordinary” general decision does not define decision classes correctly. Setting the option to TRUE makes the classifier use the transitive closure of generalized decision what eliminates the problem.

*AlphaForPartialReducts* (0.0 – 1.0) - used only if *Reducts* is set to *PartialLocal* or *PartialGlobal*. It defines $\alpha$ value in $\alpha$-cover of partial reducts.

*MissingValueDescriptorsInRules* (TRUE/FALSE) - allows or not descriptors with missing values in conditional part of rules.

### 6.2 $K$ nearest neighbours / RIONA

**Authors:** Arkadiusz Wojna, Grzegorz Góra, Lukasz Kosson

**Rseslib class path:**

rseslib.processing.classification.parameterised.knn.KnnVis

**Weka class path:**

weka.classifiers.lazy.RseslibKnn

**Description:**

$K$ nearest neighbours classifier with distance measures that can work also for data with both numeric and nominal attributes. It implements particularly fast neighbour search algorithm in large data sets. Detailed description of $k$ coefficient optimization algorithm and voting of the nearest neighbours, the analysis of metrics and attribute weighting methods can be found in [27]. The classifier has also the mode to work as RIONA algorithm [5].

The classifier is available in Weka after installing Rseslib package (see Section 7.1). One-letter options of the classifier displayed in Weka reports are explained after running (on Linux use colon, on Windows use semicolon to separate jar paths):

```
java -cp [path-to-weka.jar]:[path-to-rseslib.jar]
    weka.classifiers.lazy.RseslibKnn -h
```
The visualisation of the classifier can be seen using Qmak system (see Section 7.2). Visualization presents data spacing with respect to distance between objects. After training one can see in the visualization window that the visualising algorithm is searching iteratively for data spacing that reflects real distances. The accurate distance for any pair of objects can be displayed when the pair is selected by a user.

Parameters:

- **metric** - type of metric used for measuring distance between data objects. The distance between pair of objects is calculated as weighted linear sum of distances from all attributes:
  
  - *CityAndHamming* - combination of Manhattan metric (absolute difference between values) for numeric attributes and Hamming metric (1 if values are different, 0 if values are equal) for symbolic attributes
  
  - *CityAndSimpleValueDifference* - combination of Manhattan metric for numeric attributes with *SimpleValueDifference* metric for symbolic attributes. *SimpleValueDifference* is a kind of difference between decision distributions of a pair of attribute values in training set.
  
  - *InterpolatedValueDifference* - combination of *SimpleValueDifference* for symbolic attributes with its version for numeric attributes. The numeric version of this metric is based on dividing the range of values into intervals, counting decision distributions in the intervals from the training set and approximating decision distribution for any numeric value using linear interpolation between the two intervals nearest to a given value.
  
  - *DensityBasedValueDifference* - combination of *SimpleValueDifference* for nominal attributes with its adaptation to numerical attributes that takes into account distribution of numerical value density (does not use intervals and interpolation). Calculations of decision distribution for every numeric value is based on some neighbourhood of this value. The neighbourhood is established by the number of nearest values occurring on a given attribute in the training set. Decision distribution for a given value is calculated from a subset of training objects whose values on a given attribute fall into calculated neighbourhood.

- **vicinitySizeForDensityBasedMetric** - used only if metric = *DensityBasedValueDifference*. It defines the number of values that belong to the neighbourhood of a given numerical value defining decision distribution for this value.

- **weightingMethod** - the method of scaling distances for attributes:
  
  - *DistanceBased* - iterative correction of weights to optimize distances to correctly classified training objects
– **AccuracyBased** - iterative correction of weights to optimize training objects classification
– **Perceptron** - optimizing weights by the method of perceptron training
– **None** - using metric without scaling distances for attributes

- **indexing** (TRUE/FALSE) - if TRUE the classifier uses indexing of training objects to accelerate classification and optimization of \( k \)
- **learnOptimalK** (TRUE/FALSE) - if TRUE the classifier automatically optimizes the value of \( k \) value by optimizing classification accuracy of the training set with the leave-one-out method; if FALSE the classifier uses the value of \( k \) set by user
- **maxK** - used only if **learnOptimalK** = TRUE, defines the limit on maximal value of \( k \)
- **\( k \)** - number of nearest neighbours which take part in selection of decision for a classified object; it can be optimized automatically or set by a user
- **filterNeighboursUsingRules** - switch to RIONA which is extra filter of nearest neighbours eliminating neighbours not verified by additionally generated rules [5]
- **voting** - the method of voting for decisions by nearest neighbours:
  - **Equal** - the votes of all nearest neighbours are equally important
  - **InverseDistance** - the votes of nearest neighbours are inversely proportional to their distances from classified object
  - **InverseSquareDistance** - the votes of nearest neighbours are inversely proportional to square of their distances from classified object

### 6.3 \( K \) nearest neighbours with local metric induction

**Author:** Arkadiusz Wojna

**Rseslib class path:**
rseslib.processing.classification.parameterised.knn.LocalKnnClassifier

**Weka class path:**
weka.classifiers.lazy.LocalKnn

**Description:**
This is \( k \) nearest neighbours method extended with an extra step - the classifier calculates a local metric for each classified object [23]. While classifying a test
object, first the classifier finds a large set of the nearest neighbours (according to global metric). Then it generates a new, local metric from this large set of neighbours. At last, the \( k \) nearest neighbours are selected from this larger set of neighbours according to the locally induced metric.

In comparison to standard \( k \)-nn algorithm this method improves classification accuracy particularly for the case of data with nominal attributes [23]. It is reasonable to use this method rather for large data sets (2000+ training instances).

The classifier is available in Weka after installing Rseslib package (see Section 7.1). One-letter options of the classifier displayed in Weka reports are explained after running (on Linux use colon, on Windows use semicolon to separate jar paths):

```java
java -cp [path-to-weka.jar]:[path-to-rseslib.jar]
weka.classifiers.lazy.LocalKnn -h
```

**Parameters:**

- `metric` - as in Section 6.2
- `vicinitySizeForDensityBasedMetric` - as in Section 6.2
- `weightingMethod` - as in Section 6.2
- `learnOptimalK` - as in Section 6.2
- `localSetSize` - size of nearest neighbours set used for induction of local metric
- \( k \) - as in Section 6.2
- `voting` - as in Section 6.2

### 6.4 Decision tree C4.5

**Authors:** Arkadiusz Wojna, Maciej Próchniak

**Rseslib class path:**

`rseslib.processing.classification.tree.c45.C45Vis`

**Description:**

The implementation of C4.5 decision tree originally developed by Quinlan[16]. It enables also tree pruning after construction - automatic and manual (by user). Automatic pruning is set in trees parameters. Manual pruning is enable during visualization of the tree.
The visualization of the tree you can see using Qmak system (see Section 7.2). The visualization presents structure of the tree and information about internal nodes (description of branching and decision distribution) and leaves (decision distribution and decision). During visualization the user can change a structure of the tree (pruning of the tree) by replacing any subtree by a leaf.

The visualization of single object classification is presented as a path in the tree for this object.

**Parameters:**

- *pruning* (TRUE/FALSE) - on/off pruning after construction of the tree
- *noOfPartsForBuilding* and *noOfPartsForPruning* - the part of the training set used for construction of the tree and the part of training set used for pruning (only if pruning is on).

### 6.5 Rule classifier AQ15

**Author:** Cezary Tkaczyk

**Rseslib class path:**

rseslib.processing.classification.rules.AQ15Classifier

**Description:**

The implementation of covering rule classifier AQ15 described in [11].

**Parameters:**

- *coverage* - value from [0; 1] describes the minimal part of training set, which has to be covered by rules
- *searchWidth* - the width of rules space search while searching next (best) rule (limit of available rules set in one step; during rules generating it controls whether quality or speed is more important for user)
- *margin* - value from [0; 1] using when you define interval descriptors for numeric attributes; it describes safety level for interval defined in such descriptors
- *classificationByRuleVoting* (TRUE/FALSE) - if TRUE, the decision is voted by all rules matching to classified object; if FALSE, the classifier combines the object with decision, which is most probable for object
6.6 Neural network

Authors: Jakub Sakowicz, Damian Wójcik

Rseslib class path:
rseslib.processing.classification.neural.VisualNeuronNetwork

Description:
This implementation searches optimal weights and restarts searching always if it can not improve quality of the current network. It returns the best found network as a result. In one iteration it uses standard backward propagation algorithm[17].

The visualization of network one can see using Qmak system (see Section 7.2). The visualization presents a structure of the network and information about weights of connections between neurons. It presents also learning of the network (after setting parameter \textit{showTraining} on \textit{TRUE}), and it enables adding neurons and after that - learning the network again.

The visualization of single object classification presents strength of signals generated by neurons.

Parameters:

- \textit{timeLimit} - time limit for learning of the network (seconds)
- \textit{automaticNetworkStructure} (TRUE/FALSE) - if TRUE, the classifier creates a network with one hidden layer and set number of neurons in this layer; if FALSE, the user defines a structure of the network
- \textit{hiddenLayersSize} - used only if \textit{AutomaticNetworkStructure} = FALSE, defines number of hidden layers and number of neurons in every layer, and the sizes of hidden layers (separated by semicolons), for example the value 7;5;3 means that the network has 3 hidden layer, the first one have 7 neurons, the second one - 5 and the third one - 3 neurons
- \textit{initialAlpha} - the initial value of learning speed coefficient $\alpha$ in backward propagation algorithm. The coefficient is decreasing during the time elapsing
- \textit{targetAccuracy} - target accuracy of classification (%). If the accuracy is achieved on training data, then the learning is stopped
- \textit{showTraining} (TRUE/FALSE) - can be set on TRUE by user only using graphical interface. It means the classifier is create immediately and (after its visualizations calling) the learning-process can be watched
6.7 Naive Bayes

Author: Łukasz Ligowski

Rseslib class path:
rseslib.processing.classification.bayes.NaiveBayesClassifier

Description:
The classifier estimates conditional probability of object value for different decisions and during object classification it maximizes decision probability for given object values. It is based on Bayes theorem:
\[ P(\text{dec} = d | x) = \frac{P(x | \text{dec} = d) \cdot P(\text{dec} = d)}{P(x)} \]

Naive Bayes classifier estimates conditional probability of attributes value independently and calculates absolute conditional probability assuming independence of attributes.

For symbolic attributes, the classifier estimates probability of \( x \) value for a given decision using \( m \)-estimate:
\[ p(x | \text{dec} = d) = \frac{N_x + m}{N + m} \]
where \( N \) is the number of objects in a decision class, \( N_x \) is the number of objects in a decision class with the value \( x \) on the estimated attribute, \( Q \) is the number of possible values of estimated attribute and \( m \) is the parameter of distribution. Value \( m = 0 \) means just frequency of \( x \) occurrence in a decision class.

For numerical attributes, the classifier estimates probability of \( x \) value for a given decision using continuous distribution and kernel functions. The probability is defined using density function:
\[ p(x | \text{dec} = d) = \frac{1}{Nh} \sum_{i=1}^{N} \phi(\frac{x - x_i}{h}) \]
where \( N \) is the number of objects in a decision class, \( h \) is smoothness parameter (the greater \( h \) value means the smoother distribution), \( x_i \) is a value of a training object from a given decision class on an estimated attribute and \( \phi \) is a kernel function. There are two types of such functions:
\[ \phi(y) = \begin{cases} 1 & \text{if } |y| \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases} \] hypercube
\[ \phi(y) = \frac{1}{\sqrt{2\pi}} \left( \exp -\frac{|y|^2}{2} \right) \] Gaussian

Estimations are made after removing missing values on an estimated attribute.

Parameters:
• \textit{mEstimateParameter} - \textit{m}-estimate parameter, used for symbolic values

• \textit{kernel} - type of kernel function, used for numerical values:
  
  – \textit{gaussian}
  
  – \textit{hypercube}

• \textit{smoothness} - the parameter smoothing kernel functions, defines \( h \) in the formula for function density

### 6.8 Support vector machine

**Author:** Witold Wojtyra

**Rseslib class path:**

\texttt{rreslib.processing.classification.svm.SVM}

#### 6.8.1 Description:

Detailed description and analysis of SVM model and classifier one can see in [28]. The SVM classifier classifies only numeric data. Tuples of data are treated as vectors of \( \mathbb{R}^n \) space, where \( n \) is number of attributes. Using kernel transformations it projects data from \( \mathbb{R}^n \) to \( H \), where searching for dependencies between data is easier. Classifier training is based on finding hyper-plane in \( H \), which separates data with different values of decision attribute (this problem is solved by numeric optimization a quadratic function in this space). To enable multi-decision classification there is \( \frac{k(k-1)}{2} \) binary classifiers constructed and final decision for every tuple of data is made by voting (the winner is the class that has most votes). Besides, to make the method insensitive to noises in data, it is possible to classify with error (during training we enable that some objects are classified incorrectly - we want to get the most general classification model). The output of training stage are weights coefficients for every tuple of training data. The coefficients have non-zero values for support vectors (the name of the method is taken from this vectors). The classification phase relies on calculating decision function (it is dependent on parameters) for a new tuple of data.

#### 6.8.2 Parameters

There are parameters used in SVM classification method:

• \( C \) - describes penalty coefficient for incorrect classification. The greater it is, the more restrictive classification is. Too great value of this parameter causes slow down of the method.
• **tolerance** – coefficient of tolerance when real values are comparing. It is used during heuristic selection of points that are optimized. Too great value of this parameter causes that there are optimized too many α coefficients (the number of iterations increases) and can cause noticeable decrease of method efficiency. Too small values of this parameter makes the method sensitive to rounding errors.

• **epsilon** – the classifier stops where all input vectors satisfy Karush-Kuhn-Tucker conditions. The epsilon coefficient means acceptable error during calculation of condition value for every point. The greater coefficient, the faster the method but the smaller classification accuracy.

### 6.8.2.1 Kernel transformations

Selection of linear transformation has a large influence on classification type. Kernel transformations enable to find complex dependencies in data. The list of implemented kernel transformation (x and y means vectors from data space):

1. Linear transformation
   \[ K(x, y) = \langle x \cdot y \rangle \]

2. Polynomial transformation
   \[ K(x, y) = (\langle x \cdot y \rangle + a)^d \]
   \[ a, d \in \mathbb{R} \]

3. Gaussian transformation (RBF — Radial Basis Function)
   \[ K(x, y) = e^{-\frac{||x - y||^2}{2\sigma^2}} \]
   \[ \sigma \in \mathbb{R} \text{ - standard deviation} \]

4. Exponential transformation
   \[ K(x, y) = e^{-\frac{||x - y||}{2\sigma^2}} \]
   \[ \sigma \in \mathbb{R} \]

5. Sigmoid transformation To decipher one-letter options of the classifiers displayed in Weka reports run (on Linux):

36
1. \[ K(x, y) = \tanh(\rho \langle x \cdot y \rangle + \theta) \]

\[ \rho, \theta \in \mathbb{R} \]

### 6.8.3 Hints

Setting correct parameters is very important during classification. Especially \( C \) parameter (the greater values of this parameter, the greater penalties for incorrect classification). Setting too great value causes noticeable algorithm slow down. For more complex data it can be noticed easier, because algorithm is searching adequate separating plane with too small tolerance. It causes that for data like letters recognition, the classifier can not finish for parameters \( C \) greater than 1. For less complex data the limit is about 100. For very easy data the algorithm executes efficiently for all values of parameter \( C \). Furthermore changing the parameter does not make essential change of classification correctness. The classifier SVM achieves the best accuracy for \( C \) parameter from \( (0.05, 0.5) \).

The parameters of each kernel transformation have the greatest influence on classification accuracy. But it is hard to find a simple rule. The same parameters values causes extremely different results for different types of data, e.g. for spectro-metric data with polynomial transformation, the classifier gains very similar accuracy for any polynomial degree. But for thyroid illness data, the polynomial degree two gains noticeably worse results than polynomials of higher degree. Concluding, setting adequate parameters value is strongly related to analysed data.

### 6.9 Classifier based on principal components analysis

**Authors:** Rafał Falkowski, Łukasz Kowalski

**Rseslib class path:**

rseslib.processing.classification.parameterised.pca.PcaClassifierVis

**Description:**

Detailed description and analysis of model and classifier one can see in [3]. The classifier creates one model for every decision class.
The visualization of classifier can be seen using Qmak system (see Section 7.2). For a given class (chosen by a user) the visualization presents data projection on chosen components from model of the class. The size of the points are inversely proportional to their distances from subspaces that are models of chosen classes.

Parameters:

- \textit{principalSubspaceDim} - limit for maximal number of principal components used for classification; the number of principal components is optimized by the algorithm and is based on training set

6.10 Classifier based on local principal components analysis

Author: Rafał Falkowski

Rseslib class path:

rseslib.processing.classification.parameterised.pca.LocalPcaClassifier

Description:

Detailed description and analysis of model and classifier one can see in [3]. There are several local models created for every decision class.

Parameters:

- \textit{principalSubspaceDim} - like in Section 6.9
- \textit{noOfLocalLinearModels} - number of local models created for every decision class

6.11 Bagging

Author: Sebastian Stawicki

Rseslib class path:

rseslib.processing.classification.meta.Bagging

Description:

Metaclassifier which combines a number of “weak” classifiers to obtain one “strong” classifier proposed by Breiman [1].

Parameters:

- \textit{baggingWeakClassifiersClass} - classifier type used as “weak” classifier, given as class path
6.12 AdaBoost

Author: Sebastian Stawicki

Rseslib class path:
rseslib.processing.classification.meta.AdaBoost

Description:
Metaclassifier which combines a number of “weak” classifiers to obtain one “strong” classifier proposed by Shapire [18, 19]. The experimental results from enhancing rule classifiers from rseslib with AdaBoost method can be found in [26].

Parameters:

- adaBoostWeakClassifiersClass - classifier type used as “weak” classifier, given as class path
- adaBoostNumberOfIterations - number of iterations training “weak” classifiers
- adaBoostUseWeakClassifiersDefaultProperties (TRUE/FALSE) - if TRUE the classifier uses default parameters of “weak” classifier, if FALSE AdaBoost classifier expects that besides its parameters it is provided with all parameters of weak classifier as well.
Chapter 7

Tools

7.1 Weka

Three Rseslib classifiers with full configuration:

- Rough set based rule classifier (Section 6.1)
- K nearest neighbours / RIONA (Section 6.2)
- K nearest neighbours with local metric induction (Section 6.3)

are available as an official Weka package for testing and experimenting in Weka. The package requires Weka version 3.8.0 or newer. To install Rseslib classifiers in Weka use the following steps:

1. Download, install and start Weka GUI Chooser
2. Select Tools -> Package manager from menu
3. Press Refresh repository cache if you use already installed Weka
4. Select Rseslib from the list of available packages
5. Press Install
6. Restart Weka
7.2 Graphical user interface

Authors: Arkadiusz Wojna, Katarzyna Jachim, Damian Mański, Michał Mański, Krzysztof Mroczek, Robert Piszczatowski, Maciej Próchniak, Tomasz Romaničzuk, Piotr Skibiński, Marcin Staszczyk, Michał Szostakiewicz, Leszek Tur, Damian Wójcik, Maciej Zuchniak

Qmak is a GUI tool based on rseslib library. It is a graphical interface for data analysis, visualization of classifiers, visualization of single object classification, classifiers testing and comparison. The interface enables also testing of new classifiers implemented by a user.

To run Qmak download the latest qmak package from http://rseslib.mimuw.edu.pl, unpack the package and run the script qmak.sh (on Linux) or qmak.bat (on Windows).

You can find help on Qmak in the main menu of the application.

7.3 Computing in cluster

Author: Rafał Latkowski

Simplistic Grid Manager is a tool for running rseslib-based experiments on a cluster of computers in a network. The main features of the program:

- Capability of train-and-test experiments applied to any classifier from rseslib library
- Can be used in a network working on unreliable connections (e.g. GPRS)
- Firewall breaking system included
- UDP instead of TCP communication
- Lack of calls that can block network subystem of operating system (e.g. reverse-DNS-lookup)
- Available both on Linux and Windows

To run distributed experiment do the following steps:

1. Download the latest sgm package from http://rseslib.mimuw.edu.pl
2. Copy the package to each machine and unpack it
3. Prepare the experiment definition file (as described later) on the machine with the server
4. On one machine start the server (\texttt{sgm-server.sh} script on Linux, \texttt{sgm-server.bat} script on Windows) passing the experiment definition file as the parameter, e.g. on Linux:
\texttt{./sgm-server.sh << experiments file>>}

5. On each machine start the client (\texttt{sgm-client.sh} script on Linux, \texttt{sgm-client.bat} script on Windows) providing the server name or IP address as the parameter, e.g. on Linux:
\texttt{./sgm-client.sh << server name/address>>}

### 7.3.1 Experiment definition

To define an experiment two types of files are required on the server: the jobs file and the experiment file. Each jobs file contains the list of jobs to be executed. The server outputs one file with results for each jobs file. The experiment file provides the list of jobs files to be executed.

The structure of the experiment file is:

```
jobs filename_1 output_filename_1
jobs filename_2 output_filename_2
...
```

The structure of the jobs file is:

```
classifier_name training_file test_file param1=val1;param2=val2;...
classifier_name training_file test_file param1=val1;param2=val2;...
...
```

The following line is an example of a single job definition:

```
rseslib.processing.classification.rules.RoughSetRuleClassifier
data/att_1.trn data/att_1.tst
Discretization=EqualWidth;Reducts=PartialLocal
```

The definition of each job must be contained in a single line of the jobs file. A user does not need to specify each parameter of a classifier. If a parameter is not specified, the default value is used.

The format of the output file is the same as the jobs file except that each line describing a task is appended with the task results:

```
stat1=val1;stat2=val2;stat3=val3;...
```
7.4 Command line programs

Exemplary program executing single test (described in 5.5) for all rseslib classifiers is provided by the class `rseslib.example.TrainAndTest`. As arguments you need to pass either one data file or two data files: training and testing, e.g.:

```
java rseslib.example.TrainAndTest data/heart.dat
```

If only one argument is provided the program splits the data randomly into the training and the test part with ratio 2:1 (the header must be included into the data file).

Exemplary program executing cross-validation test (described in 5.6) for all rseslib classifiers is provided by the class `rseslib.example.CrossValidationTester`. As an arguments you need to provided the number of cross-validation folds and one data file (with the header included), e.g.:

```
java rseslib.example.CrossValidationTester 5 data/heart.dat
```

Both programs write classification results to standard output.
Bibliography


