

RSESLIB User Guide

Arkadiusz Wojna
Rafał Latkowski
Łukasz Kowalski

November 17, 2023

Contents

1	Introduction	5
2	Overview	7
2.1	Modular component-based architecture	7
2.2	Processing algorithms	8
2.3	Data-related objects	9
2.4	Library structure	10
3	Logging	12
4	Data	14
4.1	Formats	14
4.1.1	ARFF	14
4.1.2	CSV	15
4.1.3	RSES2	16
4.2	Data header	16
4.3	Data representation	17
4.4	Data table and its statistics	19
4.5	Loading and saving data	20
4.6	Vectors and value distributions	21
5	Framework for algorithms	22
5.1	Input to algorithms	22
5.2	Configuration parameters	23

5.3	Reporting progress	25
5.4	Measuring time	26
5.5	Statistics from computations	26
5.6	Saving and loading	27
6	Discretization	29
6.1	Applying discretization to data	29
6.2	Discretization types	30
6.2.1	Equal Width	30
6.2.2	Equal Frequency	31
6.2.3	One Rule	31
6.2.4	Static Entropy Minimization	32
6.2.5	Dynamic Entropy Minimization	32
6.2.6	ChiMerge	33
6.2.7	Global Maximal Discernibility Heuristic	34
6.2.8	Local Maximal Discernibility Heuristic	34
7	Discernibility matrix	35
8	Reducts	38
8.1	Reduct types	38
8.2	Reduct representation	39
8.3	Computing reducts	39
8.3.1	All global reducts	40
8.3.2	All local reducts	41
8.3.3	Johnson's reducts	41
8.3.4	Partial reducts	42
9	Rules	43
9.1	Rules representation	43
9.1.1	Rule types	43
9.1.2	Universal boolean function based rules	44
9.1.3	Optimized rules with equality descriptors	44

9.2	Generating rules	45
9.2.1	Generating rules from reducts	45
9.2.2	AQ15 algorithm	47
9.2.3	Exemplary rule generator	47
10	Classification and experiments	48
10.1	Classifiers	48
10.2	Rule-based classifiers	49
10.3	Porting Rseslib-based classifiers to Weka	50
10.4	Visualization	51
10.5	Single classifier test and classification results	51
10.6	Training and testing many classifiers	52
10.7	Crossvalidation and multiple tests	53
11	Classifier types	55
11.1	Rough set based rule classifier	55
11.2	K nearest neighbours / RIONA	58
11.3	K nearest neighbours with local metric induction	61
11.4	RIONIDA	62
11.5	Decision tree C4.5	64
11.6	Rule classifier AQ15	65
11.7	Neural network	66
11.8	Naive Bayes	67
11.9	Support vector machine	69
11.10	Classifier based on principal components analysis	72
11.11	Classifier based on local principal components analysis	73
11.12	Bagging	73
11.13	AdaBoost	74
12	WEKA	75
13	QMAK: Interaction with classifiers and their visualization	76

14 SGM: Computing many experiments on many computers/cores	78
14.1 Experiment definition & running SGM Server	79
14.2 Running SGM-Node	81
14.3 Cluster architecture and message relying	82
15 Command line programs	84
15.1 Compute and write reducts	84
15.2 Compute and write rules	85
15.3 Cross-validation on Rseslib classifiers	86
15.4 Train and test Rseslib classifiers	86
Bibliography	87

Chapter 1

Introduction

Rseslib is a library of rough set and machine learning data structures and algorithms implemented in Java. Rough set theory [18] was introduced by Pawlak as a methodology for data analysis based on approximation of concepts in information systems. Discernibility is a key concept in this methodology, which is the ability to distinguish objects, based on their attribute values. The library is not limited to rough sets, it contains and is open to concepts and algorithms from other areas of machine learning and data mining.

This user guide describes Rseslib version 3. The first version of the library started in 1993 and was implemented in C++. Rseslib 2 was the first version of the library implemented in Java and it stands for the core of Rough Set Exploration System (RSES). The version 3 was entirely redesigned and all the methods available in this version were implemented from scratch.

The two main objectives of Rseslib 3 are:

- providing open source library of algorithms and computational models from rough set theory and machine learning in Java
- providing a universal library of highly reusable and substitutable components at very elementary level unmet in other open source data mining Java libraries

The algorithms in Rseslib 3 can be used both by users who need to apply ready-to-use rough set or other machine learning methods in their data analysis tasks as well as by researchers interested in extension of the existing methods who can use the source code of the algorithms from the library as the basis for their extended implementations. The library can be used also within the following external tools:

- Weka - a popular open source machine learning software

- QMAK - a dedicated graphical interface
- Simple Grid Manager distributing computations over a network of computers

The library is distributed under GNU GPL license.

If you use Rseslib 3 in your work please cite the reference [33].

Chapter 2

Overview

2.1 Modular component-based architecture

Rseslib is designed to assure maximum reusability and substitutability of the existing components in new components of the library. Hence a strong emphasis is put on its modularity. The code is separated into loosely related elements as small as possible so that each element can be used independently of other elements. For each group of the elements of the same type a standardizing interface is defined so that each element used in an algorithm can be easily substituted by any other element of the same type. Code separation and standardization is applied both to the algorithms and to the objects.

The structure of rough set algorithms in Rseslib is one of the examples of the component-based architecture (see Figure 2.1). Each of the six modules: *Discretization*, *Logic*, *Discernibility*, *Reducts*, *Rules* and *Rough Set Classifier* provides well-abstracted algorithms with clearly defined interfaces that allow algorithms from other modules to use them as their components. For example, the algorithms computing reducts from the *Reducts* module use a discernibility matrix from the *Discernibility* module and one of the methods computing prime implicants of a CNF boolean formula from the *Logic* module. It is easy to extend each module with implementation of a new method and to add the new method as an alternative in all components using the module.

The component-based architecture of Rseslib makes it possible to implement unconventional combinations of data mining methods. For example, perceptron learning is used as one of the attribute weighting methods in the algorithm computing a distance measure between data objects. Estimation of value probability at given decision is another example of such combination: it uses k nearest neighbors voting as one of the methods defining conditional value probability.

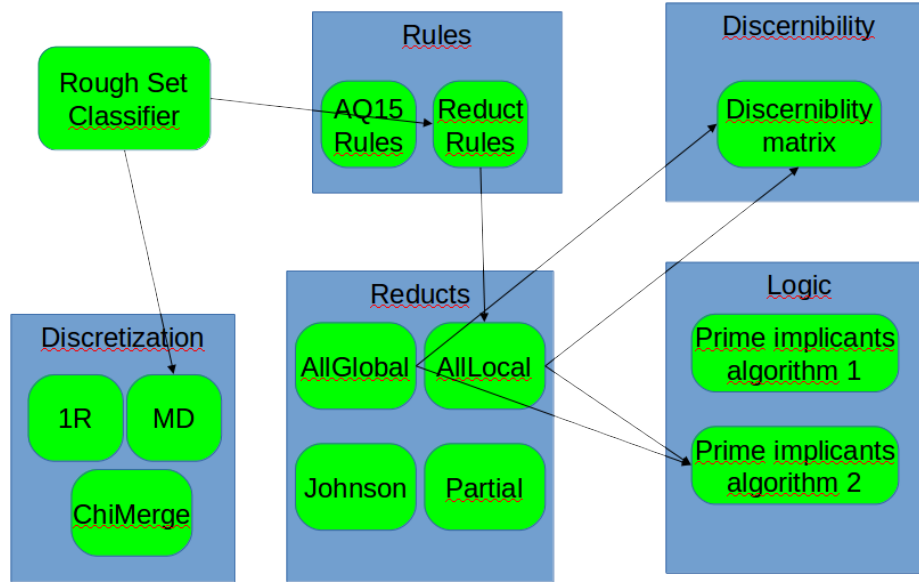


Figure 2.1: Examples of relations between Rseslib modules containing rough set algorithms

2.2 Processing algorithms

Rseslib provides many rough set and other algorithms including various classification algorithms used in machine learning and data mining. Each algorithm is available as a separate class or method and it is easy to use as an independent component. That includes:

- **Data transformation:** missing value completion (non-invasive data imputation by Gediga and Dumentsch), attribute selection, numerical attribute scaling, new attributes (radial transformation, linear transformation, arithmetic transformations)
- **Discretization:** equal width intervals, equal frequency intervals, Holte's 1R algorithm, entropy minimization (static and dynamic), ChiMerge algorithm, maximal discernibility (MD) heuristic (global and local)
- **Data filtering:** missing values filter, Wilson's editing, minimal consistent subset (MCS) by Dasarathy, universal boolean function based filter
- **Data sampling:** with repetitions, without repetitions, with given class distribution
- **Data clustering:** k approximate centers algorithm
- **Data sorting:** attribute value related, distance related

- **Discernibility matrix computation**
- **Reducts computation:** all global, all local, greedy Johnson heuristic, global partial, local partial
- **Rule induction:** from global reducts, from local reducts, AQ15 algorithm
- **Metric induction:** Hamming and Value Difference Metric (VDM) for nominal attributes, city-block Manhattan, Interpolated Value Difference Metric (IVDM) and Density-Based Value Difference Metric (DBVDM) for numerical attributes, attribute weighting (distance-based, accuracy-based, perceptron)
- **Principal Component Analysis (PCA):** OjaRLS algorithm
- **Boolean reasoning:** two different algorithms generating prime implicant from a CNF boolean formula
- **Genetic algorithm scheme:** a user provides cross-over operation, mutation operation and fitness function only
- **Classification:** rough set based rule classifier, k nearest neighbours / RIONA, k nearest neighbours with local metric induction, RIONIDA, C4.5 decision tree, AQ15 rule classifier, classical neural network, naive bayes, support vector machine, classifier based on principal component analysis (PCA), classifier based on local principal component analysis (PCA)
- **Metaclassification:** Bagging, AdaBoost
- **Classifier evaluation:** single train-and-classify test, cross-validation, multiple test with random train-and-classify split, multiple cross-validation (all types of tests can be executed on many classifiers)

2.3 Data-related objects

The algorithms listed in Section 2.2 usually transform one type of objects representing data into another type. This section enumerates briefly the types of objects in the library implementing various data-related mathematical concepts that constitute input and output of those algorithms and can be used as isolated components:

- **Basic:** attribute, data header, data object, boolean data object, numbered data object, data table, nominal attribute histogram, numeric attribute histogram, decision distribution
- **Boolean functions/operators:** attribute value equality, numerical attribute interval, nominal attribute value subset, binary discrimination, metric cube, negation, conjunction, disjunction

- **Real functions/operators:** scaling, perceptron, radius function, multiplication, addition
- **Integer functions:** discrimination (discretization, 3-value cut)
- **Decision distribution functions:** nominal value to decision distribution, numeric value to vicinity-based decision distribution, numeric value to interpolated decision distribution
- **Vector space:** vector, linear subspace, principal components subspace, vector function
- **Linear order**
- **Indiscernibility relations**
- **Discernibility matrices:** discerning all pairs of objects, pairs with different ordinary decisions, pairs with different generalized decisions, pairs with different both generalized and ordinary decisions
- **Reducts**
- **Rules:** boolean function based rule, equality descriptors rule, partial matching rule
- **Distance measures:** Hamming, Value Difference Metric (VDM), city-block Manhattan, Interpolated Value Difference Metric (IVDM), Density-Based Value Difference Metric (DBVDM), metric-based indexing tree
- **Probability:** gaussian kernel function, hypercube kernel function, m-estimate

2.4 Library structure

As written in Java Rseslib source code is divided into packages. There are eight main packages:

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

rseplib.util The package providing useful methods extending standard Java libraries.

rseslib.structure The first one of the two major packages. It covers all the structures mentioned in Section 2.3 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 implementing `java.io.Serializable` interface.

Each structure should provide also text representation by implementing the standard `toString()` method.

rseslib.processing The second one of the two major packages. It contains all data processing methods enumerated in Section 2.2. Most of them implements a certain transformation of one data representation into another one.

rseslib.qmak The package providing graphical user interface dedicated to Rseslib.

rseslib.simplegrid The package with Simple Grid Manager, a tool for running Rseslib-based experiments on many computers or cores.

rseslib.example The package providing examples of library usage from command line.

weka.classifiers The package connecting Rseslib to Weka. It contains the classes wrapping Rseslib classifiers that are recognized and provided to users by Weka tools.

Chapter 3

Logging

The main class for reporting messages and errors is the class **Report** (the `rseslib.system` package). It enables to direct control and error messages to different output channels depending on a type of application using Rseslib, for example to a console, a file or a GUI window. At the beginning each program needs to initialize the **Report** class with the type of channels used by the application. The package `rseslib.system` provides a number of predefined channels:

- **StandardOutput** is the standard output
- **StandardErrorOutput** is the standard error output
- **StandardDebugOutput** is the standard output used for debugging messages
- **FileOutput** is the channel for writing to a file

Users can define their own specific channel types. They can connect more the one channel in the same program to report messages or errors, e.g. **StandardOutput** to get information on a console and **FileOutput** to save messages in a file or any user-defined channels.

Three kinds of messages are generated by Rseslib:

Error messages Rseslib algorithms use the class **Report** to handle the error types that do not require to stop whole procedure or program, for example an incorrect parameter of a particular algorithm or an error in a single classifier while testing many algorithms or classifiers. In such situations an Rseslib procedure calls the method `Report.exception(Exception)` providing an exception and continues.

The `exception()` method reports errors to the connected channels. To connect channels for errors an application needs to call the method `Report.addErrorOutput(Output)` for each channel to be connected at the beginning of the program.

Information messages Information messages enable to communicate about the effects and the results of Rseslib algorithms. Like error messages they can be written to the console, to a file or to a graphical interface window defined by a user. The application needs to use the method `Report.addInfoOutput(Output)` at the beginning of the program to connect channels for information messages. Later it needs to use the methods `Report.display(Object)` and `Report.displaynl(Object)` providing the objects obtained as the results of Rseslib procedures. These two methods writes the results to the connected channels. They use the standard method `toString()` of the objects returned by Rseslib procedures. While extending Rseslib with new algorithms the objects representing the results are always expected to implement the `toString()` method so that the results can be handled uniformly by other users with the help of the `Report` class. 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 in the phase of development of new methods. They are used to provide control messages verifying the correctness of implemented algorithms. Like for error and information messages any number of channels can be connected using the method `Report.addDebugOutput(Output)`. Users can use the method `Report.debugnl(String)` to write debugging messages to the connected channels. The method `Report.setDebugDisplay(boolean)` turns debugging messages on and off in 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 4

Data

At present the algorithms in Rseslib are based on classical representation of data in machine learning. It is assumed that a set of objects is described by a set of conditional attributes and a decision attribute. Each object is represented by a vector of values corresponding to particular attributes. The type of a conditional attribute can be either numerical, if its values are comparable and can be represented by numbers (e.g.: age, temperature, height), or nominal, if its values are incomparable, i.e., if there is no linear order in the domain of the attribute values (e.g.: color, sex, shape). A data header describing the attribute types and data objects are the main components of data representation.

The library contains many algorithms implementing various methods of supervised learning. These methods assume that each object is assigned with a value of the decision attribute called a decision class and they learn from a training set a function approximating the real decision function on all objects outside the training set. At present the algorithms in the library assume that the domain of values of the decision attribute is discrete and finite.

4.1 Formats

Data are provided in files. The files need to contain a header describing the attribute types (columns) and data in the tabular form. Each object is represented by a single line containing the row with the attribute values. Rseslib reads 3 data formats: ARFF , CSV and RSES2.

4.1.1 ARFF

The format of the very popular open source machine learning software WEKA widely adopted in the machine learning community. The format description

is available in WEKA documentation at <http://www.cs.waikato.ac.nz/ml/weka>. Rseslib jar does not include Weka jar so to load an arff file a Weka jar must be provided in the class path while starting your program using Rseslib, e.g.:

```
java -cp ...weka.jar...
```

A user can find a Weka jar in the target directory after building Rseslib source with Maven or in Rseslib release package at <http://rseslib.mimuw.edu.pl>.

4.1.2 CSV

Comma Separated Version - a popular format that can be exchanged between databases, spreadsheet programs like Microsoft Excel or Libre Office and software recognizing this format like Rseslib.

Values in the rows of data may be separated by comma and/or whitespaces. Comments (lines starting with '#') and empty lines are allowed.

This format requires an additional header describing data. A header may be provided in two ways:

- by adding the header at the beginning of the 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 enables to use the CSV file without any extra conversion and eliminates the inconvenience of altering large files in case of very large data sets.

A 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 the 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** and **no** may be used instead of the full type name. Optionally one can add **skip** as the third keyword - then the attribute is ignored while loading the data.

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 a missing value. It means it is possible to have more than one notation

for missing value in a single data file.
The 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 with headers are available at
<http://rseslib.mimuw.edu.pl>.

4.1.3 RSES2

Format of the RSES2 system. The format description is available in the RSES2 documentation at <http://logic.mimuw.edu.pl/~rses>.

4.2 Data header

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

A data header represents a set of attributes describing data. The `Header` interface provides the basic header functionality. The method `noOfAttr()` returns the number of attributes including a decision attribute if data are provided with a decision class. The attributes in a header are indexed from 0 to `noOfAttr()-1`. The method `attribute(int i)` in the `Header` interface returns the representation of the *i*-th attribute implementing the interface `Attribute`. The name of each attribute can be obtained using the method `name()`. Attributes are categorized according to the type of their values: each attribute is either numerical (`isNumeric()` returns `true` and the attribute is represented by an object of the `NumericAttribute` class) or symbolic (`isNominal()` returns `true` and the attribute is represented by an object of the `NominalAttribute` class). Attributes are categorized also according to their roles in classification: each attribute is either conditional (`isConditional()` returns `true`), decision (`isDecision()` returns `true`) or it is ignored in classification (`isText()` returns `true`).

Data representation in `Rseslib` can handle data with many decision attributes and data with no decision attribute. In the popular case of data with exactly

one symbolic decision attribute two additional methods from the `Header` interface can be helpful. The method `decision()` returns the index of the decision attribute and the method `nominalDecisionAttribute()` provides the representation of the decision attribute.

All data structures related to a given header implement the interface `Headerable` (the `rseslib.structure` package). The interface provides the method `attributes()` that returns a header representation implementing the `Header` interface.

4.3 Data representation

The classes representing single data objects are provided in the package `rseslib.structure.data`.

The interface `DoubleDataWithDecision` extending 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. The methods `set(int attr, double val)` and `get(int attr)` are used to set and obtain attribute values, `setDecision(double val)` and `getDecision()` are used to set and obtain the decision value. Decision values can be set and obtained with the `set()` and `get()` methods like conditional attributes, the extra methods are provided just for convenience. Each data object is related to its header provided by the method `attributes()`.

The values of numerical attributes (`attributes().isNumeric(attr)==true`) are provided directly by the `get()` method, e.g.:

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

Symbolic values (`attributes().isNominal(attr)==true`) are coded in data objects with numbers. To encode and decode the values of a given symbolic attribute the methods `globalValueCode(String value)` and `stringValue(double valueCode)` of the corresponding `NominalAttribute` object are used, e.g.:

```
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));
```

The code of a symbolic value is global and it is the same across all data tables loaded in a single execution of an Rseslib-based program. It is also globally unique: two different string values have different codes even if the values are from different data tables. It means that to check whether two symbolic values are equal or not it is enough to compare their codes regardless of whether the values are from the same table or not.

In case of data with single symbolic decision the values in the `setDecision` and `getDecision` methods are global codes than can be translated to strings using the same `globalValueCode` and `stringValue` methods as for conditional symbolic attributes.

For the sake of performance many implemented algorithms use arrays whose values relate to symbolic values of a single attribute in a given data set, e.g. the value histogram. The indices of such an array represent the different values of an attribute. To facilitate the use of such arrays for each symbolic attribute Rseslib provides local mapping of its values to successive indices. Each `NominalAttribute` object has the method `noOfValues()` returning the number of different values of the attribute occurring in a data table. The symbolic values of the attribute are assigned with the local indices from 0 to `noOfValues()-1`. The methods `localValueCode(double globalValueCode)` and `globalValueCode(int localValueCode)` are used to translate global value codes into local indices and vice versa.

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

Warning! While implementing serialization of data-related objects using symbolic values the global value codes can not be serialized as they can change between different executions of Rseslib-based programs. Use one of the two other options:

- serialize original strings representing symbolic values; the global codes can be obtained with the `globalValueCode(String value)` method while deserializing the values,
- serialize the local value codes along with the `NominalAttribute` object or with the whole header; they preserve local value codes.

Warning! While implementing learning, classification or other algorithms using data objects it is important to remember that the `noOfAttr()` method includes all attributes regardless of whether an attribute is conditional, decision or to be ignored. To iterate over the attributes of a particular role a user needs to apply extra check while using the values of an object. For example, to iterate

over the values of the conditional attributes only and to skip the decision the `isConditional()` check is required:

```

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

```

4.4 Data table and its statistics

Classes for representing data tables are provided in the package `rseslib.structure.table`.

The `DoubleDataTable` interface is the main interface for representing a set of data objects. The interface provides a number of methods: the method `attributes()` returns the header of data, `add(DoubleData dData)` adds an object to a set, `remove(DoubleData dData)` removes an object from a set, `noOfObjects()` returns the number of objects in a set. The simple scheme of data table processing is:

```

    DoubleDataTable table;
    :
    for (DoubleData dData : table.getDataObjects()) {
        // do something with dData object
        :
    }

```

In case of a data table with exactly one symbolic decision attribute, the interface `DoubleDataTable` provides the decision distribution with the help of the `getDecisionDistribution()` method. It returns an array, the number of objects in the table with a given decision is provided at the position of the array equal to the local code of the decision value. The mapping between the decision values and their local codes can be obtained from the `NominalAttribute` object representing the decision attribute returned by the `attributes().nominalDecisionAttribute()` method.

If the decision attribute has two decision classes its `NominalAttribute` object provides the `getMinorityValueGlobalCode()` method that returns the decision value of the decision class that is minority in the data table.

The value distribution of conditional symbolic attributes can be obtained with the help of the method `getValueDistribution(int attr)`. Simple statistics

of numerical attributes like minimum, maximum, mean or standard deviation are provided by the method `getNumericalStatistics(int attr)`.

The `DoubleDataTable` interface has also a number of methods splitting a given table randomly into smaller tables including stratified partition.

The standard implementation of the `DoubleDataTable` interface is the class `ArrayListDoubleDataTable`. In particular, this class implements the method `toString()` providing different statistics of a table in a human-readable format including the decision distribution in case of a table with exactly one symbolic decision attribute.

4.5 Loading and saving data

To save and load data in ARFF format a Weka jar must be added to the class path (see Section 4.1.1).

To load data from a file one of `ArrayListDoubleDataTable` constructors can be used.

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

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

The second argument is an object reporting the progress on load (see Section 5.3).

To load data from a CSV file with a separate header file, first the header is loaded using the constructor `ArrayHeader(File)`, then the constructor `ArrayListDoubleDataTable(File,Header,Progress)` is used to load the data, e.g.:

```
Header hdr = new ArrayHeader(new File("data/segment.hdr"));  
DoubleDataTable table = new ArrayListDoubleDataTable(  
    new File("data/segment.trn"),  
    hdr,  
    new StdOutProgress());
```

The same header object may be used to load more than one 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 the header from a file containing both a header and data by using the same constructor `ArrayHeader(File)` as for header files.

Tables can be saved in two formats: ARFF and CSV. The saved table can be loaded again using one of the described `ArrayListDoubleDataTable` constructors. To save data in ARFF format the method `storeArff(String,File,Progress)` is used. The first argument is the name of the data set required by ARFF format. To save data in CSV format the method `store(File,Progress)` is used. The data are saved with Rseslib header added at the beginning of the file.

4.6 Vectors and value distributions

The class `Vector` from the `rseslib.structure.vector` package combines features of a value distribution and a vector. It is used to represent distributions of attribute values including decision distributions providing methods for counting values, and to represent vectors in a vector space providing a large set of operations on vectors.

Chapter 5

Framework for algorithms

5.1 Input to algorithms

There are two categories of classes implementing algorithms and computational methods in Rseslib:

- the classes that take some data-related objects as input and produce other data-related objects like the classes implementing data transformation or computation of reducts and rules
- the classes that take some data-related objects as input and represent complex models per se like classifiers

In case of the first category of classes it is enough to implement the interface dedicated to a particular type of an algorithm to be implemented. The interface defines the type of input required by a particular type of algorithms. E.g. data transformation methods need to implement the `Trasformer` interface or the more specialized `AttributeTransformer` interface.

The second category needs to implement an appropriate interface too but it is expected also to provide appropriate constructors. It is assumed that the constructors take all data structures required to build a model as arguments and after a constructor is finished the constructed object represents a ready-to-use model.

A class can provide more than one constructor depending on how much the input to the algorithm is processed. Constructors with partially precomputed input may be helpful while running repetitive experiments that do not require recalculation of whole models from scratch. Using precomputed input a user may decrease time needed to run experiments. For example, the rough set classifier class can provide a constructor taking only a data table as input,

another constructor that takes already computed reducts as input but still it needs a data table to calculate the rules and a constructor with computed rules as input:

```
public RoughSetRuleClassifier(Properties prop,
                             DoubleDataTable trainingSet) {
    // this constructor generates reducts and rules
    :
}
public RoughSetRuleClassifier(Properties prop,
                             Collection<BitSet> reducts,
                             DoubleDataTable trainingSet) {
    // this constructor generates rules
    :
}
public RoughSetRuleClassifier(Properties prop,
                             Collection<Rule> rules) {
    // this constructor stores rules only
    :
}
```

5.2 Configuration parameters

Classes implementing algorithms and computational methods expose usually configuration parameters. For example, the discretization type, the discernibility matrix type and the type of reducts are the parameters of the algorithm generating rules from reducts and the metric type and the number of neighbours are the parameters of k nearest neighbours classifier. Rseslib provides a framework for configurable classes with the class `Configuration` and it uses the standard `java.util.Properties` class to represent parameters.

While implementing a class with configuration a user needs to satisfy the following requirements:

- Provide a configuration file with default parameters values; the file must be located in the resources in the same path as the configurable class and its name must be the same as the class name with the extension *.properties*, for example:
for the class *rseslib.processing.rules.ReductRuleGenerator*
the correct path for the configuration file is
rseslib/processing/rules/ReductRuleGenerator.properties
Each line in a configuration file is either the definition of a default parameter value or a comment (beginning with the *#* character):


```

# discretization type
Discretization = EqualWidth
# number of intervals
DiscrNumberOfIntervals = 5
:

```

- Inherit from the class `Configuration` (the `rseslib.system` package) and provide a constructor with the parameters on the list of arguments. The first line of the constructor must call 1-argument constructor of the super-class passing the parameters as the argument, for example:

```

public class MyAlgorithm extends Configuration {
    public MyAlgorithm(Properties params, ...) {
        super(params);
        :
    }
    :
}

```

Describing the parameters and their possible values using comments in the configuration file is recommended good practice.

The implementation of a configurable class can use the parameter values by calling the `getProperty(String propertyName)` method. In case of boolean, integer and real-value parameters the more specialized methods: `getBoolProperty`, `getIntProperty` and `getDoubleProperty` can be used.

A user can run an algorithm providing configuration in two ways:

- by providing `null` as the parameters to the algorithm, it makes the algorithm use the default parameter values defined in the configuration file, e.g.:

```

roughCl = new RoughSetRuleClassifier(null, ...);

```

- by defining parameters in a program and passing them to the constructor, e.g.:

```

Properties props = Configuration.loadDefaultProperties(
    RoughSetRuleClassifier.class);
props.setProperty("Discretization", "EqualFrequency");
props.setProperty("DiscrNumberOfIntervals", "8");
roughCl = new RoughSetRuleClassifier(props, ...);

```

If an algorithm have many parameters and most parameters used to run the algorithm have default values a user may load the default values using the `loadDefaultProperties(Class configurableClass)` method and set only non-default values as in the example above.

5.3 Reporting progress

Computational methods that can run several seconds or more, are expected to report progress. The package `rseslib.system.progress` provides classes implementing progress reporting.

The interface `Progress` is the base interface for classes reporting progress. A constructor or a method implementing long lasting computations is expected to accept the `Progress` interface on the list of its arguments. The interface has two methods: the method `set(String name, int noOfSteps)` is called once at the beginning of computations providing a message describing computations and defining the number of steps. The method `step()` is used to notify the progress each time a subsequent step is completed. The number of steps should be defined so that notification of each step completion is easy and time of steps is as close as possible. It should not be too small to assure enough progress granulation e.g.:

```
public MyIndexingSet(..., Progress prog) {
    :
    prog.set("Indexing the data set", <<data set size>>);
    for every objects in the set
        index it
    prog.step();
}
```

If a method runs more than one component methods such that each of them can last long and each reports progress the class `MultiProgress` provides a mechanism enabling to merge the progress from all component methods into one continuous progress. To use a `MultiProgress` object a user needs to define the progress proportion between component methods. The following example defines the rough set classifier running 3 methods: discretization, computation of reducts and generation of rules and combining the progress of these 3 methods with the proportion 1:6:2:

```
public RoughSetRuleClassifier(..., Progress prog) {
    int[] proportion = { 1, 6, 2 };
    Progress combinedProg = new MultiProgress(
        "Training the rough set classifier",
        prog, proportion);

    generateDiscretization(combinedProg);
    computeReducts(combinedProg);
    generateRules(combinedProg);
}
```

For reporting progress to the standard output of a program the class `StdOutProgress` is provided. After each 2% of progress the `StdOutProgress` object writes a new

marker. To report the progress to the standard output pass the object to the method being run, e.g.:

```
indexingSet = new MyIndexingSet(..., new StdOutProgress());
```

If a user does not need any information about progress they can pass the `EmptyProgress` object as the argument.

5.4 Measuring time

The class `Timers` from the package `rseplib.util.time` provides methods for measuring time of computations. The class provides a set of timers. A user may use concurrently as many timers as they need. Each timer can be run (the `start()` method), paused (the `stop()` method) and reset (the `reset()` method) many times. Each timer provides two values. The method `getCumulatedTime()` returns the total time in milliseconds since the last reset or since the initial start of the timer (if never reset) and the method `getTime()` returns the time in milliseconds since the end of the last pause.

5.5 Statistics from computations

Algorithms can provide some statistics from computations or about constructed computational models, e.g. the number of rules generated or the number of nodes of a decision tree.

Rseplib provides a common framework for handling such statistics with the class `ConfigurationWithStatistics` (the package `rseplib.system`). If a user wishes to provide statistics in a given class implementing an algorithm the class must inherit from the `ConfigurationWithStatistics` class and implement the methods `calculateStatistics()` and `resetStatistics()`. The `calculateStatistics()` method calls the method `addToStatistics(String name, String value)` for each computed statistics, e.g.:

```
int noOfReducts, noOfRules;
:
void calculateStatistics() {
    addToStatistics("Number of reducts",
                    Integer.toString(noOfReducts));
    addToStatistics("Number of rules",
                    Integer.toString(noOfRules));
}
```

Statistics can be collected over many calls to an object representing a computational model, e.g. they can be collected in a classifier from many calls of the `classify()` method. The method `resetStatistics()` is used to notify the object to reset statistics calculation.

The statistics can be obtained from objects providing statistics by calling the method `getStatistics()`.

5.6 Saving and loading

To enable saving data-related objects and computed models to a file and reloading them later the classes representing them are expected to implement the standard `java.io.Serializable` interface.

Warning! To avoid enforcing implementation of the `java.io.Serializable` interface by all configurable classes with statistics (extending the `ConfigurationWithStatistics` class), the `ConfigurationWithStatistics` class does not implement this interface, but provides the serialization methods: `writeConfigurationAndStatistics(ObjectOutputStream)` and `readConfigurationAndStatistics(ObjectInputStream)`. While implementing serialization in a configurable class with statistics, a user needs always to implement the methods from the `Serializable` interface explicitly calling the `writeConfigurationAndStatistics` and `readConfigurationAndStatistics` methods inside.

Warning! By analogy, while implementing serialization in a configurable class without statistics (extending the `Configuration` class) a user needs always to implement the methods from the `Serializable` interface explicitly calling the `writeConfiguration` and `readConfiguration` methods inside.

To save a serializable object in a file use the standard `java.io.ObjectOutputStream` class, e.g.:

```
RoughSetRuleClassifier roughCl;
:
File file = new File("obj/my_rough.cl");
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream(file));
out.writeObject(cl);
out.close();
```

To reload a saved object from a file use the standard `java.io.ObjectInputStream` class, e.g.:

```
File file = new File("obj/my_rough.cl");
```

```
ObjectInputStream in = new ObjectInputStream(  
    new FileInputStream(file));  
Classifier cl = (Classifier)in.readObject();  
in.close();
```

Chapter 6

Discretization

Discretization (known also as quantization, binning or bucketing) is data transformation that converts data from numeric attributes into nominal attributes. Some algorithms require data in the form of nominal attributes, e.g. some rule based algorithms like the rough set based classifier.

Rseslib provides a number of discretization methods. Each method computes a number of cuts for each numerical attribute from a given data set. The cuts define a number of disjoint intervals. The numerical values from each interval are represented by a single nominal value in a discretized attribute.

Interfaces and classes for discretizations are provided in the packages `rseslib.processing.discretization` and `rseslib.processing.transformation`.

6.1 Applying discretization to data

Each discretization is assumed to implement the interface `Transformer` from the `rseslib.processing.transformation` package. The interface provides the method `transformToNew(DoubleData dObj)` that returns the discretized representation of an input data object.

If a user needs to discretize the whole data table that can be done using the static method `transform(DoubleDataTable table, Transformer transformation)` from the `TableTransformer` class (the package `rseslib.processing.transformation`). The header of a discretized table is different from the header of the original table: all attributes in the discretized table are nominal. The nominal values of discretized attributes are described by value ranges of the original numerical attributes.

The methods generating discretizations implement the interface `TransformationProvider` (the package `rseslib.processing.transformation`).

This interface provides the method `generateTransformer(DoubleDataTable table)`. The method returns a generated discretization of the type `Transformer`.

The following example discretizes a table using the local Maximal Discernibility heuristic method:

```
DoubleDataTable table;
:
TransformationProvider method = new MDLocalDiscretizationProvider();
Transformer discretization = method.generateTransformer(table);
DoubleDataTable discretizedTable =
    TableTransformer.transform(table, discretization);
```

Some discretization methods have parameters. The constructor of the classes implementing such methods takes the parameter values as its argument (see Section 5.2). Providing `null` as the parameters makes a parameterized method use its default parameter values.

6.2 Discretization types

More detailed description and experimental comparison of available discretization methods can be found in [9].

6.2.1 Equal Width

Author: Rafał Latkowski

Class path:

`rreslib.processing.discretization.RangeDiscretizationProvider`

Description:

The range of values of a numerical attribute in a data set is divided into k intervals of equal length. The number of intervals k is the parameter of the method.

Parameters:

- *DiscrNumberOfIntervals* - the number of intervals for each numerical attribute.

6.2.2 Equal Frequency

Author: Rafał Latkowski

Class path:

`rseslib.processing.discretization.HistogramDiscretizationProvider`

Description:

The range of values of a numerical attribute in a data set is divided into k intervals containing the same number of objects from a data set. The number of objects in particular intervals may differ by one if the size of the data set does not divide by k . The number of intervals k is the parameter of the method.

Parameters:

- *DiscrNumberOfIntervals* - the number of intervals for each numerical attribute.

6.2.3 One Rule

Author: Marcin Jajmężna

Class path:

`rseslib.processing.discretization.OneRuleDiscretizationProvider`

Description:

Holte's 1R algorithm [8] tries to cut the range of values of a numerical attribute into intervals containing training objects with the same decision but it avoids very small intervals. The minimal number n of training objects that must fall into each interval is the parameter of 1R algorithm. The algorithm executes the following steps:

1. Sort the objects by the values of a numerical attribute to be discretized
2. Scan the objects in the ascending order adding them to an interval until one of the decision classes, denote it by d , has n representatives in the interval
3. While the decision of the object next in the ascending order is d add the object to the interval
4. Start the next interval as empty and go to 2

Parameters:

- *DiscrMinimalFrequency* - the minimal number of data objects that must fall into each interval generated by 1R algorithm.

6.2.4 Static Entropy Minimization

Author: Marcin Jałmużna

Class path:

`rseslib.processing.discretization.EntropyMinStaticDiscretizationProvider`

Description:

Static entropy minimization [5] is a top-down local method discretizing a single numerical attribute. It starts with the whole range of values of the attribute in a data set and divides it into smaller intervals. At each step the algorithm remembers which objects from the data set fall into each interval. In a single step the algorithm searches all possible cuts in all intervals and selects the new cut c maximizing information gain, i.e. minimizing entropy:

$$E(a_i, c, S) = \frac{|S_1|}{|S|} Ent(S_1) + \frac{|S_2|}{|S|} Ent(S_2) \quad (6.1)$$

where

$$Ent(S) = - \sum_{j=1}^m \frac{|\{x \in S : dec(x) = d_j\}|}{|S|} \log \left(\frac{|\{x \in S : dec(x) = d_j\}|}{|S|} \right)$$

a_i is the attribute to be discretized, S is the set of the objects falling into the interval on a_i containing a candidate cut c , $S_1 = \{x \in S : x_i \leq c\}$, $S_2 = \{x \in S : x_i > c\}$.

The method applies the minimum description length principle to decide when to stop the algorithm.

6.2.5 Dynamic Entropy Minimization

Author: Marcin Jałmużna

Class path:

`rseslib.processing.discretization.EntropyMinDynamicDiscretizationProvider`

Description:

Dynamic entropy minimization method [5] is similar to static entropy minimization but it discretizes all numerical attributes at once. It starts with the whole set of objects and splits it into two subsets with the optimal cut selected from all numerical attributes. Then the algorithm splits each subset recursively scanning all possible cuts over all numerical attributes at each split. To select the best cut the algorithm minimizes the same formula 6.1 as the static method.

On average the dynamic method is faster than the static method and produces fewer cuts.

6.2.6 ChiMerge

Author: Marcin Jałmużna

Class path:

`rreslib.processing.discretization.ChiMergeDiscretizationProvider`

Description:

ChiMerge [11] is a bottom-up discretization method using χ^2 statistics to test whether neighbouring intervals have significantly different decision distributions. If the distributions are similar the algorithm merges the intervals into one interval. The method discretizes each numerical attribute independently.

The method has two parameters. The first parameter n is the minimal number of final intervals. The second parameter is the confidence level (0.0 – 1.0) used to recognize two neighbouring intervals as different and not to merge them.

First, the algorithm calculates the threshold θ from χ^2 distribution with $m - 1$ degrees of freedom and a given confidence level and starts with a separate interval for each value of a numerical attribute occurring in a data set U . At each step it merges the pair of neighbouring intervals with the minimal χ^2 value as long as this minimal value is less than θ and the number of intervals does not drop below n . χ^2 value is defined as:

$$\chi^2(S_1, S_2) = \sum_{j=1}^m \frac{(|S_1^j| - ES_1^j)^2}{ES_1^j} + \sum_{j=1}^m \frac{(|S_2^j| - ES_2^j)^2}{ES_2^j}$$

where S_1, S_2 are the sets of objects from U falling into two neighbouring intervals, $S_k^j = \{x \in S_k : dec(x) = d_j\}$ and ES_k^j is the expected number of objects in S_k with the decision d_j :

$$ES_k^j = |S_k| \frac{|\{x \in S_1 \cup S_2 : dec(x) = d_j\}|}{|S_1 \cup S_2|}$$

Parameters:

- *DiscrConfidenceLevelForIntervalDifference* (0.0–1.0) - the confidence level required to consider two neighbouring intervals as different and not to merge them by *ChiMerge* method.
- *DiscrMinimalNumberOfIntervals* - the minimal number of intervals for each numerical attribute generated by *ChiMerge* method.

6.2.7 Global Maximal Discernibility Heuristic

Author: Marcin Jałmużna

Class path:

`rreslib.processing.discretization.MDGlobalDiscretizationProvider`

Description:

Global maximal discernibility heuristic method [15] is a top-down dynamic method discretizing all numerical attributes at once. At each step it evaluates cuts globally with respect to the whole training set. It starts with the set S^* of all pairs of objects with different decisions defined as:

$$S^* = \{\{x, y\} \subseteq U : dec(x) \neq dec(y)\}$$

At each step the algorithm finds the cut c that discerns the greatest number of pairs in the current set S^* , adds the cut c to the result set and removes all pairs discerned by the cut c from the set S^* . The optimal cut is searched among all possible cuts on all numerical attributes. The algorithm stops when the set S^* is empty.

6.2.8 Local Maximal Discernibility Heuristic

Author: Marcin Jałmużna

Class path:

`rreslib.processing.discretization.MDLocalDiscretizationProvider`

Description:

Local maximal discernibility heuristic method [15] selects the cuts optimizing the number of pairs of discerned objects like the global method but the procedure selecting the best cut is applied recursively to the subsets of objects obtained by splitting the data set by the previously selected cuts.

It starts with the best cut for the whole training set U splitting it into subsets U_1 and U_2 . Next the discretization algorithm selects the best cut splitting U_1 and recursively the best cuts with respect to the subsets of U_1 . Next it searches independently for the best cuts for U_2 . At each step the best cut is searched over all attributes.

Chapter 7

Discernibility matrix

Author: Rafał Latkowski

Class path:

`rseslib.processing.discernibility.DiscernibilityMatrixProvider`

Description:

The library provides 4 types of discernibility matrix [26] including types handling inconsistencies in data [19, 25]. Let U be a data table used to construct a discernibility matrix and A be the set of conditional attributes describing U . Each type is $|U| \times |U|$ matrix defined for all pairs of objects $x, y \in U$. The fields of discernibility matrix $M(x, y)$ are defined as the subsets of the set of conditional attributes: $M(x, y) \subseteq A$. If a data table contains numerical attributes discernibility matrix can be computed using either the original or the discretized numerical attributes.

The first type of discernibility matrix M^{all} depends on the values of the conditional attributes only, it does not take the decision attribute into account:

$$M^{all}(x, y) = \{a_i \in A : x_i \neq y_i\}$$

In many applications, e.g. in object classification, we need to discern objects only if they have different decisions. The second type of discernibility matrix M^{dec} discerns objects from different decision classes:

$$M^{dec}(x, y) = \begin{cases} \{a_i \in A : x_i \neq y_i\} & \text{if } dec(x) \neq dec(y) \\ \emptyset & \text{if } dec(x) = dec(y) \end{cases}$$

If data are inconsistent, i.e. if there are one or more pairs of objects with different decisions and with equal values on all conditional attributes:

$$\exists x, y \in U : \forall a_i \in A : x_i = y_i \wedge dec(x) \neq dec(y)$$

then $M^{dec}(x, y) = \emptyset$ like for pairs of objects from the same decision class. To overcome this inconsistency the concept of generalized decision was introduced [19, 25]:

$$\partial(x) = \{d \in V_{dec} : \exists y \in U : \forall a_i \in A : x_i = y_i \wedge dec(y) = d\}$$

If U contains inconsistent objects x, y they have the same generalized decision. The next type of discernibility matrix M^{gen} is based on generalized decision:

$$M^{gen}(x, y) = \begin{cases} \{a_i \in A : x_i \neq y_i\} & \text{if } \partial(x) \neq \partial(y) \\ \emptyset & \text{if } \partial(x) = \partial(y) \end{cases}$$

This type of discernibility matrix removes inconsistencies but discerns pairs of objects with the same original decision, e.g. an inconsistent object from a consistent object. The fourth type of discernibility matrix M^{both} discerns a pair of objects only if they have both the original and the generalized decision different:

$$M^{both}(x, y) = \begin{cases} \{a_i \in A : x_i \neq y_i\} & \text{if } \partial(x) \neq \partial(y) \wedge dec(x) \neq dec(y) \\ \emptyset & \text{if } \partial(x) = \partial(y) \vee dec(x) = dec(y) \end{cases}$$

Data can contain missing values. All types of discernibility matrix available in the library have 3 modes to handle missing values [12, 28]:

- different value — an attribute a_i discerns x, y if the value of one of them on a_i is defined and the value of the second one is missing (missing value is treated as yet another value): $a_i \notin M(x, y) \Leftrightarrow x_i = y_i \vee (x_i = * \wedge y_i = *)$
- symmetric similarity — an attribute a_i does not discern x, y if the value of any of them on a_i is missing: $a_i \notin M(x, y) \Leftrightarrow x_i = y_i \vee x_i = * \vee y_i = *$
- nonsymmetric similarity — asymmetric discernibility relation between x and y : $a_i \notin M(x, y) \Leftrightarrow (x_i = y_i \wedge y_i \neq *) \vee x_i = *$

The first mode treating missing value as yet another value keeps indiscernibility relation transitive but the next two modes make it intransitive. Such a relation is not an equivalence relation and does not define correctly indiscernibility classes in the set U . To eliminate that problem the library provides an option to transitively close an intransitive indiscernibility relation.

Usage:

To compute a discernibility matrix for a given data table a user needs to create a `DiscernibilityMatrixProvider` object passing parameters and the table to the constructor `DiscernibilityMatrixProvider(Properties prop,`

`DoubleDataTable table`). Providing `null` as the parameters makes the algorithm use the default parameter values. Section 5.2 describes how to prepare non-default parameter values. The object provides two methods.

The method `getDiscernibilityMatrix()` returns the discernibility matrix for the data table passed to the constructor. The matrix is represented by an object of the `Collection<BitSet>` type. Each `BitSet` element of the result represents the set $M(x, y)$ of conditional attributes discerning a certain pair of objects x, y from the data table. The method `get(int i)` of a `BitSet` representing $M(x, y)$ returns `true` if the i -th attribute discerns the objects x, y otherwise it returns `false`. The attribute indices are defined by the header of the data table used to compute the matrix.

The method `getLocalDiscernibility(DoubleData object)` returns the row of the discernibility matrix with the sets of attributes discerning a given data object from other objects from the data table. The row is represented by an object of the `Collection<BitSet>` type by analogy to the whole discernibility matrix.

Parameters:

- *IndiscernibilityForMissing* - defines how missing values are treated while discerning objects:
 - *DiscernFromValue* - missing value as different value
 - *DontDiscernFromValue* - symmetric similarity
 - *DiscernFromValueOneWay* - nonsymmetric similarity
- *DiscernibilityMethod* - defines the type of discernibility matrix to be computed:
 - *OrdinaryAndInconsistenciesOmitted* - the discernibility matrix type M^{dec} , it discerns objects with different decisions.
 - *GeneralizedDecision* - the discernibility matrix type M^{gen} , it discerns objects with different generalized decisions.
 - *GeneralizedDecisionAndOrdinaryChecked* - the discernibility matrix type M^{both} , it discerns objects having both generalized and original decisions different.
 - *All* - the discernibility matrix type M^{all} , it discerns all objects except for indiscernible pairs.
- *GeneralizedDecisionTransitiveClosure* (TRUE/FALSE) - is used only if *DiscernibilityMethod* is set to *GeneralizedDecision* or *GeneralizedDecisionAndOrdinaryChecked* and *IndiscernibilityForMissing* is set to *DontDiscernFromValue* or *DiscernFromValueOneWay*. Setting the option to TRUE makes the discernibility matrix use the transitive closure of generalized decision.

Chapter 8

Reducts

8.1 Reduct types

Reduct [26] is a key concept in rough set theory. It can be used to remove some data without loss of information or to generate decision rules. Let U be a data table, A be the set of conditional attributes describing U and M be a discernibility matrix of any type computed from the data table U .

Definition 1. *The subset of attributes $R \subseteq A$ is a (global) reduct in relation to a discernibility matrix M if each pair of objects discernible by M is discerned by at least one attribute from R and no proper subset of R holds that property:*

$$\begin{aligned} \forall x, y \in U : M(x, y) \neq \emptyset \Rightarrow R \cap M(x, y) \neq \emptyset \\ \forall R' \subsetneq R \exists x, y \in U : M(x, y) \neq \emptyset \wedge R' \cap M(x, y) = \emptyset \end{aligned}$$

If M is a decision-dependent discernibility matrix the reducts related to M are the reducts related to the decision attribute.

Reducts defined in Definition 1 called also global reducts are sometimes too large and generate too specific rules. To overcome this problem the notion of local reducts was introduced [35].

Definition 2. *The subset of attributes $R \subseteq A$ is a local reduct in relation to a discernibility matrix M and an object $x \in U$ if each object $y \in U$ discerned from x by M is discerned from x by at least one attribute from R and no proper subset of R holds that property:*

$$\begin{aligned} \forall y \in U : M(x, y) \neq \emptyset \Rightarrow R \cap M(x, y) \neq \emptyset \\ \forall R' \subsetneq R \exists y \in U : M(x, y) \neq \emptyset \wedge R' \cap M(x, y) = \emptyset \end{aligned}$$

It may happen that local reducts are still too large. In the extreme situation there is only one global or local reduct equal to the whole set of attributes A . In such situations partial reducts [14, 16] can be helpful.

Let P be the set of all pairs of objects $x, y \in U$ discerned by a discernibility matrix M : $P = \{\{x, y\} \subseteq U : M(x, y) \neq \emptyset\}$ and let $\alpha \in (0; 1)$.

Definition 3. *The subset of attributes $R \subseteq A$ is a global α -reduct in relation to a discernibility matrix M if it discerns at least $(1 - \alpha) |P|$ pairs of objects discernible by M and no proper subset of R holds that property:*

$$\begin{aligned} |\{\{x, y\} \subseteq U : R \cap M(x, y) \neq \emptyset\}| &\geq (1 - \alpha) |P| \\ \forall R' \subsetneq R : |\{\{x, y\} \subseteq U : R' \cap M(x, y) \neq \emptyset\}| &< (1 - \alpha) |P| \end{aligned}$$

Let $P(x)$ be the set of all objects $y \in U$ discerned from $x \in U$ by a discernibility matrix M : $P(x) = \{y \in U : M(x, y) \neq \emptyset\}$ and let $\alpha \in (0; 1)$.

Definition 4. *The subset of attributes $R \subseteq A$ is a local α -reduct in relation to a discernibility matrix M and an object $x \in U$ if it discerns at least $(1 - \alpha) |P(x)|$ objects discernible from x by M and no proper subset of R holds that property:*

$$\begin{aligned} |\{y \in U : R \cap M(x, y) \neq \emptyset\}| &\geq (1 - \alpha) |P(x)| \\ \forall R' \subsetneq R : |\{y \in U : R' \cap M(x, y) \neq \emptyset\}| &< (1 - \alpha) |P(x)| \end{aligned}$$

8.2 Reduct representation

Each reduct is represented by an object of the `java.util.BitSet` type. The method `get(int i)` of a `BitSet` representing a reduct returns `true` if the i -th attribute belongs to the reduct otherwise it returns `false`. The attribute indices are defined by the header of the data table (see Section 4.2) used to compute the reducts.

8.3 Computing reducts

The library provides a number of algorithms computing reducts. These algorithms are the most time-consuming among rough set algorithms, the time cost of other steps in the overall knowledge discovery process is often negligible when compared to reduct computations. The performance of the methods computing reducts implemented in Rseslib measured on benchmark data sets is presented in [33].

Interfaces and classes for methods computing reducts are provided in the package `rseplib.processing.reducts`.

There are two basic interfaces for methods computing reducts: the interface `GlobalReductsProvider` for methods computing global reducts and the interface `LocalReductsProvider` for methods computing local reducts. A class computing reducts can implement one of the two interfaces or both.

Each class computing reducts is required to extend the abstract `Configuration` class and to provide the constructor with the following two arguments:

```
public MyReductProvider(Properties params, DoubleDataTable table)
{
    super(params);
    :
}
```

The first argument `params` defines the parameters of the method. Providing `null` as the parameters makes the method use the default parameter values. Section 5.2 describes how to prepare non-default parameter values. The data table provided as the second argument is used to compute reducts.

The `GlobalReductsProvider` interface provides the method `getReducts()` returning `Collection<BitSet>` representing computed global reducts as described in Section 8.2. The `LocalReductsProvider` interface provides the method `getSingleObjectReducts(DoubleData object)` returning `Collection<BitSet>` representing local reducts computed for a given data object.

8.3.1 All global reducts

Author: Rafał Latkowski, Michał Kurzydłowski

Class path:

`rseplib.processing.reducts.AllGlobalReductsProvider`

Description:

The algorithm computes all global reducts from a data set. The algorithm is based on the fact that a set of attributes is a reduct if and only if it is a prime implicant of a boolean CNF formula generated from the discernibility matrix [24]. In the first phase the algorithm computes the discernibility matrix of the type specified by its parameters (see Chapter 7). In the second phase it transforms the discernibility matrix into a boolean CNF formula and applies an efficient algorithm finding all prime implicants of the formula. The algorithm finding all prime implicants uses well-known in the field of boolean reasoning

advanced techniques accelerating computations [2]. All found prime implicants are global reducts.

Parameters:

- *IndiscernibilityForMissing* - as in Chapter 7
- *DiscernibilityMethod* - as in Chapter 7
- *GeneralizedDecisionTransitiveClosure* - as in Chapter 7

8.3.2 All local reducts

Author: Rafał Latkowski, Michał Kurzydłowski

Class path:

`rseslib.processing.reducts.AllLocalReductsProvider`

Description:

The algorithm computes all local reducts for each object in a data set [33]. Like the algorithm computing global reducts it uses boolean reasoning. The first step is the same as for global reducts: the discernibility matrix specified by parameters is calculated. Next for each object x in the data set the row of the discernibility matrix corresponding to the object x is transformed into a CNF formula and all local reducts for the object x are computed with the algorithm finding prime implicants.

Parameters:

- *IndiscernibilityForMissing* - as in Chapter 7
- *DiscernibilityMethod* - as in Chapter 7
- *GeneralizedDecisionTransitiveClosure* - as in Chapter 7

8.3.3 Johnson's reducts

Author: Wiktor Gromniak

Class path:

`rseslib.processing.reducts.JohnsonReductsProvider`

Description:

The algorithm computes reducts using Johnson's greedy heuristic [10, 17]. The computed reducts are global reducts. The algorithm provides two modes.

In the first mode the algorithm computes one reduct. Like the algorithm computing all reducts first it calculates a discernibility matrix. Next the algorithm starts with the empty set of attributes as a candidate set for a reduct. At each step the algorithm considers only the non-empty fields of the discernibility matrix that do not contain any of the attributes added to the candidate set previously. In a single step it adds the attribute that occurs in the maximal number of such fields. When each non-empty field of the discernibility matrix is covered by at least one attribute from the candidate set the last part of the algorithm is checking which attributes in the candidate set can be removed. The final candidate set is a reduct. Ties in the steps selecting an attribute are resolved arbitrarily.

The second mode is the version of Johnson's greedy algorithm in which the algorithm branches and traverses all possibilities rather than selecting one of them arbitrarily when more than one attribute cover the maximal number of uncovered fields of the discernibility matrix. The result is the set of the reducts found in all branches of the algorithm.

Parameters:

- *IndiscernibilityForMissing* - as in Chapter 7
- *DiscernibilityMethod* - as in Chapter 7
- *GeneralizedDecisionTransitiveClosure* - as in Chapter 7
- *Reducts* - mode of the algorithm:
 - *OneJohnson* - computing one arbitrary Johnson's reduct
 - *AllJohnson* - computing all Johnson's reducts

8.3.4 Partial reducts

Authors: Marcin Piliszczuk, Beata Zielosko

Class path:

`rreslib.processing.reducts.PartialReductsProvider`

Description:

The class provides two greedy algorithms computing partial reducts described in [14]. The method `getReducts()` returns one global α -reduct (see Definition 3 in Section 8.1) for a given data table. The method `getSingleObjectReducts(DoubleData object)` returns one local α -reduct (see Definition 4 in Section 8.1) for given data object and data table.

Parameters:

- *AlphaForPartialReducts* (0.0 – 1.0) - α value in α -cover of partial reducts

Chapter 9

Rules

9.1 Rules representation

Classes for rules are defined in the package `rseplib.structure.rule`.

9.1.1 Rule types

There are four interfaces defining different types of rules:

- **Rule** - the basic interface for rules with deterministic decision returned by the `getDecision()` method, the interface has the method `matches(DoubleData dObj)` of boolean result type determining whether a data object matches a rule or not
- **RuleWithStatistics** - an interface extending the **Rule** interface for the rules with information about accuracy (the `getAccuracy()` method) and support (the `getSupport()` method)
- **DistributedDecisionRule** - an interface for the rules with non-deterministic decision using decision probabilities (the `getDecisionVector()` method) rather than pointing to a single decision
- **PartialMatchingRule** - an interface for the rules that can be matched partially by data objects, the method `matchesPartial(DoubleData dObj)` returns the level of matching from the range $[0; 1]$.

Each class representing rules provides text representation by implementing the standard `toString()` method.

9.1.2 Universal boolean function based rules

The `BooleanFunctionRule` class enables to define rules that can match data objects with any kind of boolean function. It provides the constructor `BooleanFunctionRule(BooleanFunction premise, double decision, NominalAttribute decAttr)`. The `premise` argument is any boolean function defining whether data objects match a rule or not, `decision` is the decision of the rule and `decAttr` is the structure describing the decision attribute.

The `BooleanFunction` interface is defined in the package `rslib.structure.function.booleanval`. The package includes implementation of the most popular boolean functions and operators used for rule construction:

- `AttributeEquality` - defines a single descriptor that requires a particular value on a given attribute
- `AttributeInterval` - defines a single descriptor that requires the value of a numerical attribute to fall into a given interval
- `AttributeValueSubset` - defines a single descriptor that requires the value of a symbolic attribute to be in a given subset of attribute values
- `Conjunction`, `Disjunction` - make conjunction and disjunction of two or more boolean functions
- `Negation` - negates a given boolean function

To construct rules a user can use these boolean functions and operators and/or implement their own functions. Below there is an example of construction of a typical rule with equality descriptors:

```
DoubleDataWithDecision obj;
:
BooleanFunction[] descr = new descriptors[3];
descr[0] = new AttributeEquality(1, obj.get(1));
descr[1] = new AttributeEquality(4, obj.get(4));
descr[2] = new AttributeEquality(7, obj.get(7));
Rule r = new BooleanFunctionRule(new Conjunction(descr),
                                obj.getDecision());
```

9.1.3 Optimized rules with equality descriptors

For the sake of performance and low memory footprint the rules with equality descriptors only have the optimized implementation by the class `EqualityDescriptorsRule`. The class provides the constructor `EqualityDescriptorsRule(BitSet descriptors,`

`DoubleData object`) taking the bit mask of the attributes having the descriptors in the constructed rule and the data object defining the values of the selected attributes in the descriptors of the rule.

The `EqualityDescriptorsRule` class implements all four types of rules described in Section 9.1.1.

9.2 Generating rules

Interfaces and classes for methods generating rules are provided in the package `rseslib.processing.rules`.

The interface `RuleGenerator` is used to define algorithms generating rules. Rules are generated from a data table with the method `generate(DoubleDataTable tab, Progress prog)` returning a collection of generated rules.

If a method generating rules has parameters it provides one-argument constructor with parameter values as the argument (see Section 5.2).

9.2.1 Generating rules from reducts

Author: Rafał Latkowski

Class path:

`rseslib.processing.rules.ReductRuleGenerator`

Description:

The algorithm generating rules from reducts [33] is based on the methods computing reducts (see Chapter 8). It differs a little depending on whether the reducts used to generate the rules are global or local.

While using global reducts first the algorithm computes the global reducts according to parameters. Let U be a data table used to generate reducts and rules and GR be the set of global reducts computed from U . In the next phase the algorithm finds all templates in the data table:

$$Templates(GR) = \left\{ \bigwedge_{a_i \in R} a_i = x_i : R \in GR, x \in U \right\}$$

Then, for each template the algorithm generates one rule with a non-deterministic decision:

$$Rules(GR) = \{t \Rightarrow (p_1, \dots, p_m) : t \in Templates(GR)\}$$

where the decision probabilities p_j in each rule $t \Rightarrow (p_1, \dots, p_m)$ are defined as:

$$p_j = \frac{|\{x \in U : x \text{ matches } t \wedge dec(x) = d_j\}|}{|\{x \in U : x \text{ matches } t\}|}$$

In case of local reducts let $LR : U \mapsto \mathcal{P}(A)$ be the algorithm computing the local reducts $LR(x)$ for each data object $x \in U$ according to parameters. First, the algorithm LR is applied to each object $x \in U$ to generate local reducts. Next, the set of templates is computed as union of the sets of templates from all objects in U :

$$Templates(LR) = \left\{ \bigwedge_{a_i \in R} a_i = x_i : R \in LR(x), x \in U \right\}$$

The set of decision rules is obtained from the set of templates in the same way as in case of global reducts:

$$Rules(LR) = \{t \Rightarrow (p_1, \dots, p_m) : t \in Templates(LR)\}$$

where p_j is defined like in the rules generated from global reducts.

The algorithm provides the option to allow the values in the descriptors of a rule to be missing values: $a_i = *$. An object x satisfies a descriptor with missing value $a_i = *$ if the value of the attribute a_i on x is missing.

The type of rules generated from reducts is always `EqualityDescriptorsRule`. As the interface method returns a collection of objects of the basic `Rule` type, to obtain accuracy and support of the generated rules the rules need to be cast to the `RuleWithStatistics` type and to obtain the probabilities of decisions the rules need to be cast to the `DistributedDecisionRule` type.

Warning! When using global reducts the algorithm does not report progress until global reducts are computed. It reports progress only while generating rules from the computed global reducts. For the case with all global reducts the computation time can be estimated by running first the algorithm computing rules from all local reducts which reports progress steadily. Computation time of all global reducts is comparable (usually a bit shorter) to computation time of all local reducts.

Parameters:

- *Reducts* - method generating reducts from discernibility matrix:
 - *AllLocal* - generating all local reducts (see Section 8.3.2)
 - *AllGlobal* - generating all global reducts (see Section 8.3.1)
 - *OneJohnson* - generating one reduct by greedy Johnson's algorithm (see Section 8.3.3)
 - *AllJohnson* - generating all reducts that may be obtained from Johnson's algorithm (see Section 8.3.3)
 - *PartialLocal* - generating local partial reducts (see Section 8.3.4)
 - *PartialGlobal* - generating global partial reducts (see Section 8.3.4)

- *IndiscernibilityForMissing* - as in Chapter 7
- *DiscernibilityMethod* - as in Chapter 7
- *GeneralizedDecisionTransitiveClosure* - as in Chapter 7
- *AlphaForPartialReducts* - as in Section 8.3.4
- *MissingValueDescriptorsInRules* (TRUE/FALSE) - allows or not descriptors with missing values in the conditional part of rules

9.2.2 AQ15 algorithm

Author: Cezary Tkaczyk

Class path:

`rseslib.processing.rules.CoveringRuleGenerator`

Description:

Implementation of the covering algorithm AQ15 generating rules described in [13]. The type of the rules generated by this algorithm is `BooleanFunctionRule`.

Parameters:

- *coverage* - value from $[0; 1]$ defines the minimal part of the data 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 generation it controls whether quality or speed is more important for a user)
- *margin* - value from $[0; 1]$ used to define interval descriptors for numeric attributes; it describes safety level for intervals defined in such descriptors

9.2.3 Exemplary rule generator

Author: Arkadiusz Wojna

Class path:

`rseslib.processing.rules.AccurateRuleGenerator`

Description:

A simple exemplary implementation of rule generation. For a given data table it generates the set of maximally specific rules containing equality descriptors for all conditional attributes, one rule for each data object. The generated rules are of the `BooleanFunctionRule` type.

Parameters:

- *maxNumberOfRules* - exemplary parameter, the limit on the number of generated rules

Chapter 10

Classification and experiments

Interfaces and classes for classifiers and experiments with classifiers are provided in the package `rseplib.processing.classification`.

10.1 Classifiers

There are two basic interfaces for classifiers: the `Classifier` interface for classifiers with deterministic decision and the `ClassifierWithDistributedDecision` interface for classifiers providing decision probabilities. A typical classifier implements one of these two interfaces and extends the abstract `ConfigurationWithStatistics` class:

```
public MyClassifier extends ConfigurationWithStatistics
                        implements Classifier {
    :
}
```

Each classifier is required to provide the constructor with the following three arguments:

```
public MyClassifier(Properties params,
                    DoubleDataTable trainTable, Progress prog)
{
    super(params);
    :
}
```

The constructor is assumed to train the classifier with a given training table and a given set of parameters. Providing `null` as the parameters makes the classifier use the default parameter values. Section 5.2 describes how to prepare non-default parameter values. This constructor is mandatory to make the classifier usable in the methods implementing experiments and in the graphical interface. The constructed classifier is ready for classification.

Classifier implementation may provide other constructors, e.g. with partially precomputed components (see 5.1).

A classifier needs to implement also a classification method. The `Classifier` interface defines the method `classify(DoubleData)` returning a decision assigned to a classified data object. In case of data with symbolic decision the result is assumed to be the global code of the assigned decision value (see Section 4.3). The `ClassifierWithDistributedDecision` interface is dedicated to data with symbolic decision only. It defines the method `classifyWithDistributedDecision(DoubleData)` returning a decision distribution assigned to a classified data object. The result is an array, the value at the i -th position of the array defines the probability of the decision value whose local code is equal to i (see Section 4.3).

Warning! While implementing a classification method a classifier is expected to handle correctly the values of conditional symbolic attributes not occurring in a training set. In particular, the object representing a symbolic attribute in the training set may not have any local code representing such a value.

To provide statistics from learning or classification a classifier needs to implement the `calculateStatistics()` and `resetStatistics()` methods (see Section 5.5).

To enable saving and reloading a classifier needs to implement the `java.io.Serializable` interface (see Section 5.6).

The types of classifiers implemented in Rseslib are described in Chapter 11.

10.2 Rule-based classifiers

Rule-based classifiers are provided in the package `rseslib.processing.classification.rules`. All the general principles of classifier implementation described in the previous section apply also to rule-based classifiers.

A typical rule-based classifier uses an existing or newly implemented generator of rules in the training phase (see Section 9.2), e.g.:

```
public MyRuleClassifier extends ConfigurationWithStatistics
    implements Classifier {
```

```

Collection<Rule> m_Rules;

public MyRuleClassifier(Properties params,
                        DoubleDataTable trainTable, Progress prog) {
    super(params);
    RuleGenerator gen;
    :
    m_Rules = gen.generate(trainTable);
}
:
}

```

Then the generated rules are used in a classification method to determine a decision of a data object to be classified. The `generate()` method returns the objects representing rules of the most general `Rule` type. To use more specific attributes of generated rules the classification method needs to cast the rules to a required subtype given the rules implement the subtype, e.g. to use rule support the rules need to be cast to the `RuleWithStatistics` type.

Rseslib provides the following classifiers based on rules: `RoughSetRuleClassifier` (Section 11.1), `AQ15Classifier` (Section 11.6) and the simple `MajorityClassifierWithRules` to be used as a training example.

10.3 Porting Rseslib-based classifiers to Weka

To enable an Rseslib-based classifier in Weka the author does not need to implement translation of data and classification results between Rseslib and Weka. Such translation is implemented in Rseslib in a universal way so that the author of any Rseslib-based classifier can use the implemented translation to provide its port to Weka.

To provide the Weka port the author needs to define a separate class extending the abstract class `AbstractRseslibClassifierWrapper` from the package `weka.classifiers`. The porting class needs to implement the constructor passing the class of the underlying Rseslib classifier to the abstract wrapper and the single-line `main()` method, e.g.:

```

public MyWekaClassifier extends AbstractRseslibClassifierWrapper {

    public MyWekaClassifier() {
        super(MyRseslibClassifier.class);
    }
    :
}

```

```

        public static void main(String[] args) throws Exception {
            runClassifier(new MyWekaClassifier(), args);
        }
    }
}

```

The class implementing the port must be placed in the appropriate subpackage of the `weka.classifiers.*` package to make it accessible within Weka tools.

If a classifier to be ported provides configuration parameters the class implementing porting must implement the configuration-related methods required by Weka tools (see Section *Writing a new Classifier* in WEKA Manual provided within each Weka installation).

It is good to provide also a short description of a classifier displayed by Weka tools by implementing the `globalInfo()` method.

The classes `weka.classifiers.rules.RoughSet` or `weka.classifiers.lazy.RseslibKnn` are the examples of Weka ports of Rseslib-based classifiers.

10.4 Visualization

Classifiers can implement graphical presentation. Such classifiers can be displayed using QMAK program (see Chapter 13).

The `VisualClassifier` interface extending the `Classifier` interface is used for classifiers implementing visualization. It requires two additional methods to be implemented. The method `draw(JPanel canvas)` draws the classifier on a graphical panel. The method `drawClassify(JPanel canvas, DoubleData obj)` presents classification of a single data object.

To make the non-visual part of visualized classifiers clear and simple visualization is usually implemented by a subclass of the class implementing the non-visual version of a classifier. For example, the class `RoughSetRuleClassifier` provides the non-visual version of rough set based classifier and the class `VisualRoughSetClassifier` is its extension implementing visualization.

10.5 Single classifier test and classification results

To run a simple experiment testing a trained classifier against a data table the class `SingleClassifierTest` can be used. The method `classify(Classifier cl, DoubleDataTable testTable, Progress prog)` returns a `TestResult` object containing the classification results. The method `getNoOfObject(double trueDec, double assignedDec)` of a `TestResult` object provides the confusion matrix: for a given decision class `trueDec` it returns the number of

the tested objects classified as `assignedDec`. The method `getAccuracy()` returns overall classification accuracy for the whole test table. The method `getDecAccuracy(double dec)` returns the classification accuracy in a given decision class. Accuracy is the ratio of correctly classified objects to all objects. If the decision attribute has two decision classes the method `getGmean()` returns G-mean classification measure, and the methods `getFmeasure()`, `getSensitivity()`, `getSpecificity()` and `getPrecision()` return F-measure, sensitivity (recall, true positive rate), specificity (true negative rate) and precision (positive predictive value) calculated for the minority decision class. The method `toString()` prints overall accuracy, accuracy in each decision class and statistics calculated by a classifier during classification. In case of data with two decision classes it prints also the values of G-mean measure, F-measure and sensitivity.

10.6 Training and testing many classifiers

The class `ClassifierSet` facilitates testing many classifiers against the same pair of the train and the test tables.

First, a user needs to define the set of tested classifiers by calling the `addClassifier(String name, Class classifierType, Properties params)` method for each instance of a classifier to be tested. The argument `name` is an arbitrary name provided by a user to identify the classifier results, the argument `params` defines the parameters of the classifier (see Section 5.2). The two-argument method `addClassifier(String name, Class classifierType)` can be used to add classifiers with default parameter values. More than one instance of same classifier type may be added to the set, e.g. with different parameter values.

After defining the set of classifiers to be tested a user trains the classifiers with the method `train(DoubleDataTable, Progress)`. At last, they test the trained classifiers with the method `classify(DoubleDataTable, Progress)`. The following code illustrates an exemplary test of 2 classifiers:

```
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 the mapping between the names of the tested classifiers (passed as the arguments to the `addClassifier(...)` method) and the

classification results represented by `TestResult` objects (see Section 10.5). The mapping can be displayed using the method of the `Report` class as below given at least one output channel is added for information messages (see Chapter 3), e.g.:

```
Report.displayMapWithMultiLines("Classification results", results)
```

Both `train(...)` and `classify(...)` methods can be used many times on the same `ClassifierSet` instance. When the `train(...)` method is run again, the classifiers are retrained with a new training table.

The `train(...)` method uses the 3-argument constructor described in Section 10.1 to train classifiers. A user may add a classifier trained with another constructor with the use of the method `addClassifier(String name, Classifier cl)`. The classifiers added with this method are skipped in the `train(...)` method. They are used only in the `classify(...)` method to provide classification results.

10.7 Crossvalidation and multiple tests

The library provides also tools for more complex tests: cross-validation (the class `CrossValidationTest`), multiple test with random partition of a test table (the class `MultipleRandomSplitTest`) and multiple cross-validation (the class `MultipleCrossValidationTest`). A typical usage scheme for all these classes is similar: first an object representing a selected test type is constructed given a set of classifiers to be tested (see Section 10.6), then the test method is run, e.g.:

```
DoubleDataTable testTab;
ClassifierSet mySet;
:
Properties cvParams;
cvParams.setProperty("noOfFolds", "10");
CrossValidationTest cvt = new CrossValidationTest(cvParams, mySet);
Map<String, MultipleTestResult> results =
    cvt.test(testTab, new StdOutProgress());
```

The two other multiple test types are run in the same way. The result returned by the test methods is the mapping between classifier names defined in the `ClassifierSet` object and their classification results. The classification result for a single classifier is represented by an object of the `MultipleTestResult` class. A `MultipleTestResult` object provides the average accuracy from all test

runs (the `getAvgAccuracy()` method) and the standard deviation of accuracy (the `getAccuracyStandardDeviation()` method). In case of data with two decision classes it provides also G-mean measure (the `getAvgGmean()` method), and F-measure (the `getAvgFmeasure()` method) and sensitivity (the `getAvgSensitivity()` method) calculated for the minority class. The measures can be obtained also in a text form using the `toString()` method. The results for all test classifiers can be displayed using the method of the `Report` class as below given at least one output channel is added for information messages (see Chapter 3), e.g.:

```
Report.displayMapWithMultiLines("Test results", results)
```

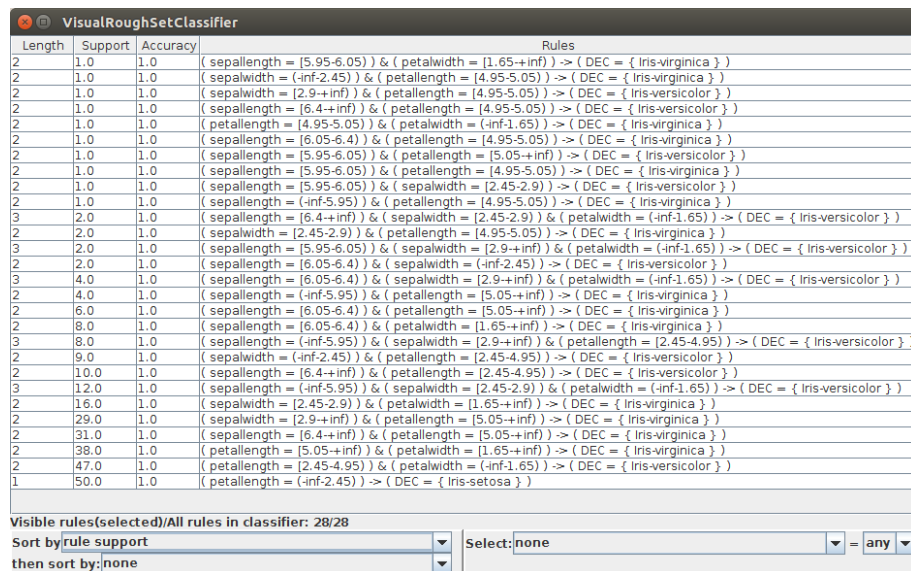
The parameters of all tests are provided using the configuration mechanism described in Section 5.2. The `CrossValidationTest` class has the one parameter `noOfFolds` defining the number of cross-validation folds.

The `MultipleCrossValidationTest` class has the two parameters: `noOfFolds` and `noOfTests` defining the number of cross-validation tests to be run. The `MultipleRandomSplitTest` class has three parameters: `noOfTests` and the pair of parameters `noOfPartsForTraining` and `noOfPartsForTesting` defining the ratio with which an input table is split randomly into the training and the test table in each test.

Chapter 11

Classifier types

11.1 Rough set based rule classifier



Length	Support	Accuracy	Rules
2	1.0	1.0	(sepalength = [5.95-6.05]) & (petalwidth = [1.65-+inf]) -> (DEC = { Iris-virginica })
2	1.0	1.0	(sepalwidth = (-inf-2.45]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-virginica })
2	1.0	1.0	(sepalwidth = [2.9-+inf]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-versicolor })
2	1.0	1.0	(sepalength = [6.4-+inf]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-versicolor })
2	1.0	1.0	(petallength = [4.95-5.05]) & (petalwidth = (-inf-1.65]) -> (DEC = { Iris-virginica })
2	1.0	1.0	(sepalength = [6.05-6.4]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-virginica })
2	1.0	1.0	(sepalength = [5.95-6.05]) & (petallength = [5.05-+inf]) -> (DEC = { Iris-versicolor })
2	1.0	1.0	(sepalength = [5.95-6.05]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-virginica })
2	1.0	1.0	(sepalength = [5.95-6.05]) & (sepalwidth = [2.45-2.9]) -> (DEC = { Iris-versicolor })
2	1.0	1.0	(sepalength = (-inf-5.95]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-virginica })
3	2.0	1.0	(sepalength = [6.4-+inf]) & (sepalwidth = [2.45-2.9]) & (petalwidth = (-inf-1.65]) -> (DEC = { Iris-versicolor })
2	2.0	1.0	(sepalwidth = [2.45-2.9]) & (petallength = [4.95-5.05]) -> (DEC = { Iris-virginica })
3	2.0	1.0	(sepalength = [5.95-6.05]) & (sepalwidth = [2.9-+inf]) & (petalwidth = (-inf-1.65]) -> (DEC = { Iris-versicolor })
2	2.0	1.0	(sepalength = [6.05-6.4]) & (sepalwidth = (-inf-2.45]) -> (DEC = { Iris-versicolor })
3	4.0	1.0	(sepalength = [6.05-6.4]) & (sepalwidth = [2.9-+inf]) & (petalwidth = (-inf-1.65]) -> (DEC = { Iris-versicolor })
2	4.0	1.0	(sepalength = (-inf-5.95]) & (petallength = [5.05-+inf]) -> (DEC = { Iris-virginica })
2	6.0	1.0	(sepalength = [6.05-6.4]) & (petallength = [5.05-+inf]) -> (DEC = { Iris-virginica })
2	8.0	1.0	(sepalength = [6.05-6.4]) & (petalwidth = [1.65-+inf]) -> (DEC = { Iris-virginica })
3	8.0	1.0	(sepalength = (-inf-5.95]) & (sepalwidth = [2.9-+inf]) & (petallength = [2.45-4.95]) -> (DEC = { Iris-versicolor })
2	9.0	1.0	(sepalwidth = (-inf-2.45]) & (petallength = [2.45-4.95]) -> (DEC = { Iris-versicolor })
2	10.0	1.0	(sepalength = [6.4-+inf]) & (petallength = [2.45-4.95]) -> (DEC = { Iris-versicolor })
3	12.0	1.0	(sepalength = (-inf-5.95]) & (sepalwidth = [2.45-2.9]) & (petalwidth = (-inf-1.65]) -> (DEC = { Iris-versicolor })
2	16.0	1.0	(sepalwidth = [2.45-2.9]) & (petalwidth = [1.65-+inf]) -> (DEC = { Iris-virginica })
2	29.0	1.0	(sepalwidth = [2.9-+inf]) & (petallength = [5.05-+inf]) -> (DEC = { Iris-virginica })
2	31.0	1.0	(sepalength = [6.4-+inf]) & (petallength = [5.05-+inf]) -> (DEC = { Iris-virginica })
2	38.0	1.0	(petallength = [5.05-+inf]) & (petalwidth = [1.65-+inf]) -> (DEC = { Iris-virginica })
2	47.0	1.0	(petallength = [2.45-4.95]) & (petalwidth = (-inf-1.65]) -> (DEC = { Iris-versicolor })
1	50.0	1.0	(petallength = (-inf-2.45]) -> (DEC = { Iris-setosa })

Visible rules(selected)/All rules in classifier: 28/28

Sort by:rule support Select:none = any

then sort by:none

Authors: Rafał Latkowski, Krzysztof Niemkiewicz

Rseslib class path:

`rseslib.processing.classification.rules.roughset.RoughSetRuleClassifier`

Weka class path:

`weka.classifiers.rules.RoughSet`

Description:

Rough set classifier [33] uses the algorithms computing discernibility matrix, reducts and rules generated from reducts described in the previous chapters. It enables to apply any of the discretization methods described in Section 6.2 to transform numerical attributes into nominal attributes. Using discernibility matrix described in Chapter 7 the classifier provides modes to work with incomplete data (with missing values) and with inconsistent data. A user of the classifier selects a discretization method, a type of discernibility matrix and an algorithm generating reducts. The classifier computes a set of decision rules with non-deterministic decision (see Section 9.2.1) and the support of each rule in the training set.

Let U be a training set and $Rules$ be the computed set of decision rules. The rules are used in classification to determine a decision value when provided with an object x to be classified. First, the classifier calculates the vote of each decision class $d_j \in V_{dec}$ for the object x :

$$vote_j(x) = \sum_{\{t \Rightarrow (p_1, \dots, p_m) \in Rules : x \text{ matches } t\}} |\{y \in U : y \text{ matches } t \wedge dec(y) = d_j\}|$$

Then the classifier assigns to the object x the decision with the greatest vote:

$$dec_{roughset}(x) = \max_{d_j \in V_{dec}} vote_j(x)$$

The classifier provides the method `getRules()` returning the set of generated rules used for classification. The type of rules used in the classifier is `EqualityDescriptorsRule` (see Section 9.1.3). As the `getRules()` method returns a collection of objects of the basic `Rule` type, to obtain accuracy and support of the generated rules the rules need to be cast to the `RuleWithStatistics` type and to obtain the probabilities of decisions the rules need to be cast to the `DistributedDecisionRule` type.

The classifier is available in Weka after installing Rseslib package (see Chapter 12). 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
```

The classifier can be visualized using *QMAK* program (see Chapter 13). Visualization of the classifier presents all the generated decision rules with their length, support and accuracy. The rules can be filtered and sorted by attribute occurrence, attribute values, rule length, support and accuracy.

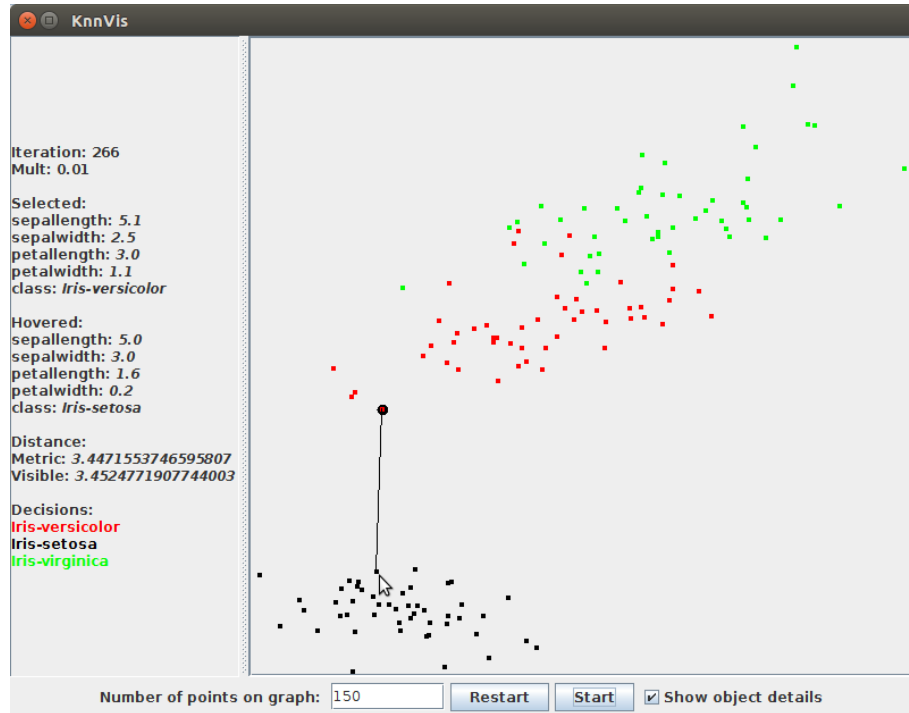
Visualization of single object classification presents the decision rules matching the classified object enabling the same types of filtering and sorting criteria as visualization of the classifier.

Rough set classifier can be stored in a file (see Section 5.6).

Parameters:

- *Discretization* - discretization method applied to numerical attributes:
 - *None* - the classifier does not use discretization
 - *EqualWidth* - equal width intervals described in Section 6.2.1
 - *EqualFrequency* - equal frequency intervals described in Section 6.2.2
 - *OneRule* - Holte's 1R algorithm described in Section 6.2.3
 - *EntropyMinimizationStatic* - static entropy minimization described in Section 6.2.4
 - *EntropyMinimizationDynamic* - dynamic entropy minimization described in Section 6.2.5
 - *ChiMerge* - ChiMerge algorithm described in Section 6.2.6
 - *MaximalDiscernibilityHeuristicGlobal* - global maximal discernibility (MD) heuristic described in Section 6.2.7
 - *MaximalDiscernibilityHeuristicLocal* - local maximal discernibility (MD) heuristic described in Section 6.2.8
- *DiscrNumberOfIntervals* - used only if *Discretization* is set to *EqualWidth* or *EqualFrequency*, see Sections 6.2.1 and 6.2.2
- *DiscrMinimalFrequency* - used only if *Discretization* is set to *OneRule*, see Section 6.2.3
- *DiscrConfidenceLevelForIntervalDifference* (0.0 – 1.0) - used only if *Discretization* is set to *ChiMerge*, see Section 6.2.6
- *DiscrMinimalNumberOfIntervals* - used only if *Discretization* is set to *ChiMerge*, see Section 6.2.6
- *Reducts* - as in Section 9.2.1
- *IndiscernibilityForMissing* - as in Chapter 7
- *DiscernibilityMethod* - as in Chapter 7
- *GeneralizedDecisionTransitiveClosure* - as in Chapter 7
- *AlphaForPartialReducts* - as in Section 8.3.4
- *MissingValueDescriptorsInRules* (TRUE/FALSE) - as in Section 9.2.1

11.2 K nearest neighbours / RIONA



Authors: Arkadiusz Wojna, Grzegorz Góra, Łukasz Kosson

Rseslib class path:

`rseslib.processing.classification.parameterised.knn.KnnClassifier`

Weka class path:

`weka.classifiers.lazy.RseslibKnn`

Description:

K nearest neighbours classifier providing various distance measures working also for data with both numerical and nominal attributes [31]. The classifier provides also a number of methods adjusting attribute weights in the distance measures, an algorithm based on leave-one-out test finding automatically the optimal number of nearest neighbours and various methods of voting by nearest neighbours. Detailed description of the classifier and all its algorithms and the experimental comparison of the distance measures, the attribute weighting algorithms and the voting methods can be found in [31]. The classifier has also the mode to work as RIONA algorithm [6, 7].

The classifier implements fast nearest neighbour search using a metric tree with two search pruning criteria [30, 31]. The implemented search method makes the classifier work for very large data sets.

The classifier is available in Weka after installing Rseslib package (see Chapter 12). 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 classifier can be visualized using *QMAK* program (see Chapter 13). Visualization of a k-nn classifier projects all training objects on 2-dimensional area of the classifier window using different colors for different decisions. It displays the process of searching for placement of the objects reflecting best the true distances between them in the induced metric. A user can select one object and hover another one to display their attribute values and the true distance between them.

Visualization of single object classification by a k-nn classifier projects the object to be classified and its k nearest neighbors also trying to reflect the true distances between them. As in model visualization neighbors from different decision classes have different colors and *QMAK* shows their attribute values and the distances between them.

The classifier can be stored in a file (see Section 5.6).

Parameters:

- *metric* - type of metric used for measuring distance between data objects. The distance between pair of objects is calculated as weighted sum of distances over all attributes:
 - *CityAndHamming* - combination of city-block Manhattan metric (absolute difference between values) for numerical attributes and Hamming metric (1 if values are different, 0 if values are equal) for symbolic attributes
 - *CityAndSimpleValueDifference* - combination of city-block Manhattan metric for numerical attributes with *Value Difference Metric (VDM)* for symbolic attributes. *Value Difference Metric* considers two symbolic values to be similar if they correlate similarly with the decision in a training set.
 - *InterpolatedValueDifference* - combination of *Value Difference Metric* for symbolic attributes with its version for numerical attributes. The numeric version of this metric is based on dividing the range of values into intervals, counting the decision distributions in the intervals from the training set and approximating decision distribution for each numeric value using linear interpolation between the two intervals nearest to a given value.

- *DensityBasedValueDifference* - combination of *Value Difference Metric* for symbolic attributes with its adaptation to numerical attributes that takes into account density of attribute values. Decision distribution for each numerical value is computed in some neighbourhood of this value. The neighbourhood of a value of a numerical attribute is defined as the set of fixed cardinality containing the training objects with the nearest values on this attribute.
- *vicinitySizeForDensityBasedMetric* - used only if *metric = DensityBasedValueDifference*. It defines the number of training objects that belong to the neighbourhood of a given numerical value and determine decision distribution for this value.
- *weightingMethod* - the method of scaling distances for attributes:
 - *DistanceBased* - iterative correction of attribute weights minimizing distances of nearest neighbors that classify correctly in a training set
 - *AccuracyBased* - iterative correction of attribute weights optimizing leave-one-out classification accuracy in a training set
 - *Perceptron* - optimization of attribute weights by perceptron training
 - *None* - using a metric without attribute weights
- *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 searches for the best value of k value by optimizing the leave-one-out classification accuracy in a training set; if FALSE the classifier uses the value of k set by a user
- *maxK* - used only if *learnOptimalK = TRUE*, defines the range in which the classifier searches for the best 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* (TRUE/FALSE) - switch to RIONA which excludes from voting the nearest neighbours not confirmed by additionally generated rules [6, 7]
- *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 a classified object
 - *InverseSquareDistance* - the votes of nearest neighbours are inversely proportional to square of their distances from a classified object

11.3 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 [27]. 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 [27]. 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 Chapter 12). 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):

The classifier can be stored in a file (see Section 5.6).

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

Parameters:

- *metric* - as in Section 11.2
- *vicinitySizeForDensityBasedMetric* - as in Section 11.2
- *weightingMethod* - as in Section 11.2
- *learnOptimalK* - as in Section 11.2
- *localSetSize* - size of nearest neighbours set used for induction of local metric
- *k* - as in Section 11.2
- *voting* - as in Section 11.2

11.4 RIONIDA

Author: Grzegorz Góra

Rseslib class path:

`rseslib.processing.classification.parameterised.knn.rionida.RIONIDA`

Weka class path:

`weka.classifiers.lazy.RIONIDA`

Description:

RIONIDA is an extension of RIONA classifier (see Section 11.2) dedicated to imbalanced data, working only for data with two decision classes. By analogy to RIONA it combines instance-based learning with rule induction but it contains substantial modifications and extensions aimed at achieving high classification quality in case of imbalanced data. Detailed description of the classifier and all its algorithms and the experimental results can be found in [6].

The classifier adds new parameters that enable to treat the minority decision in a special way and to specify to what extent the minority decision is more important than the majority decision. RIONIDA provides also more flexible mechanism than RIONA for inclusion of rules into decision selection: new parameters control the impact of rules on this process. Different parametrisations correspond to different approaches, including a pure instance-based approach, a pure rule-based approach, and combination of both.

In the learning phase RIONIDA searches the space of values of three (optionally four) parameters and selects the optimal combination of values. A user can select a measure to be maximized more relevant for imbalanced data like F-measure and G-mean.

The classifier is available in Weka after installing Rseslib package (see Chapter 12). 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.RIONIDA -h
```

The classifier can be stored in a file (see Section 5.6).

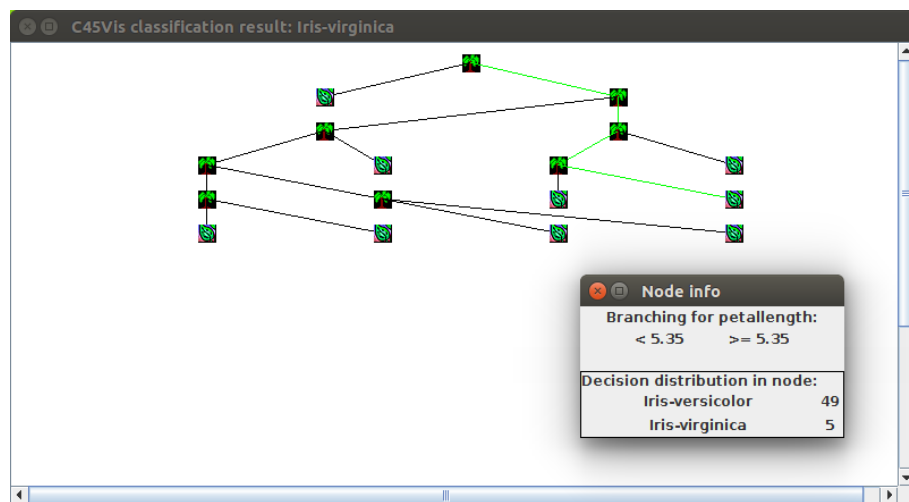
Parameters:

- *useMajorityDecAsMinorityDec* (TRUE/FALSE) - switch indicating whether the classifier treats the majority decision as the minority decision
- *metric* - as in Section 11.2

- *vicinitySizeForDensityBasedMetric* - as in Section 11.2
- *weightingMethod* - as in Section 11.2
- *indexing* (TRUE/FALSE) - if TRUE the classifier uses indexing of training objects to accelerate classification and optimization of parameters
- *learnOptimalParameters* (TRUE/FALSE) - if TRUE the classifier searches for the best combination of parameters using the leave-one-out classification in a training set; if FALSE the classifier uses the values of parameters set by a user
- *optimisation4D* - if FALSE the classifier optimizes the parameters *k*, *pThreshold* and *sMinority*; if TRUE the classifier optimizes also *sMajority*
- *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
- *maxK* - used only if *learnOptimalParameters* = TRUE, the maximal possible *k* while learning the optimal value
- *pThreshold* - if weight of minority decision divided by sum of weights is greater than or equal to this threshold then a test object is classified with minority decision, otherwise it is classified with majority decision; it can be optimized automatically or set by a user (this threshold is called *pValue* in the source code of RIONIDA)
- *pThresholdMin* - the minimal possible value while learning the optimal *pThreshold* (used only if *learnOptimalParameters* = TRUE)
- *pThresholdMax* - the maximal possible value while learning the optimal *pThreshold* (used only if *learnOptimalParameters* = TRUE)
- *pThresholdStep* - the density of values between *pThresholdMin* and *pThresholdMax* while learning the optimal *pThreshold* (used only if *learnOptimalParameters* = TRUE)
- *sMinority* - consistency level for minority decision; it can be optimized automatically or set by a user (the parameter is called *sMinorityValue* in the source code of RIONIDA)
- *sMinorityMin* - the minimal possible value while learning the optimal *sMinority* value (used only if *learnOptimalParameters* = TRUE)
- *sMinorityMax* - the maximal possible value while learning the optimal *sMinority* value (used only if *learnOptimalParameters* = TRUE)
- *sMinorityStep* - the density of values between *sMinorityMin* and *sMinorityMax* while learning the optimal *sMinority* value (used only if *learnOptimalParameters* = TRUE)

- *sMajority* - consistency level for majority decision; it can be optimized automatically or set by a user (the parameter is called *sMajorityValue* in the source code of RIONIDA)
- *sMajorityMin* - the minimal possible value while learning the optimal *sMajority* value (used only if *learnOptimalParameters* = TRUE and *optimisation4D* = TRUE)
- *sMajorityMax* - the maximal possible value while learning the optimal *sMajority* value (used only if *learnOptimalParameters* = TRUE and *optimisation4D* = TRUE)
- *sMajorityStep* - the density of values between *sMajorityMin* and *sMajorityMax* while learning the optimal *sMajority* value (used only if *learnOptimalParameters* = TRUE and *optimisation4D* = TRUE)
- *optimisationMeasure* - the measure used for optimization of the parameters:
 - *Gmean* - G-mean classification measure
 - *Fmeasure* - F-measure for minority decision
 - *Accuracy* - classification accuracy
- *filterNeighboursUsingRules* (TRUE/FALSE) - switch indicating whether rules are included in the process of decision selection
- *voting* - as in Section 11.2

11.5 Decision tree C4.5



Authors: Arkadiusz Wojna, Maciej Próchniak

Rseslib class path:

`rseslib.processing.classification.tree.c45.C45`

Description:

The implementation of C4.5 decision tree originally developed by Quinlan[20].

The classifier enables tree pruning after construction. That can be done automatically by the training algorithm or manually by a user. Use of the algorithmic pruning is defined by a parameter of the classifier. Manual pruning can be done by a user while visualizing a tree.

The tree can be visualized using *QMAK* program (see Chapter 13). Visualization of the classifier presents the structure of the tree. After selection of a node the decision distribution of its training objects is displayed and the branching condition for an inner node or the assigned decision for a leaf. A user can cut off the subtree of any inner node and convert the node to a leaf (tree pruning).

Visualization of single object classification presents a decision tree with the path from the root to a leaf corresponding to a classified object highlighted in green.

The classifier can be stored in a file (see Section 5.6).

Parameters:

- *pruning* (TRUE/FALSE) - if TRUE the algorithmic tree pruning is applied after construction of a tree
- *noOfPartsForBuilding* and *noOfPartsForPruning* - ratio used to split a training set into the part used for tree construction and the part used for pruning (used only if *pruning* is TRUE).

11.6 Rule classifier AQ15

Author: Cezary Tkaczyk

Rseslib class path:

`rseslib.processing.classification.rules.AQ15Classifier`

Description:

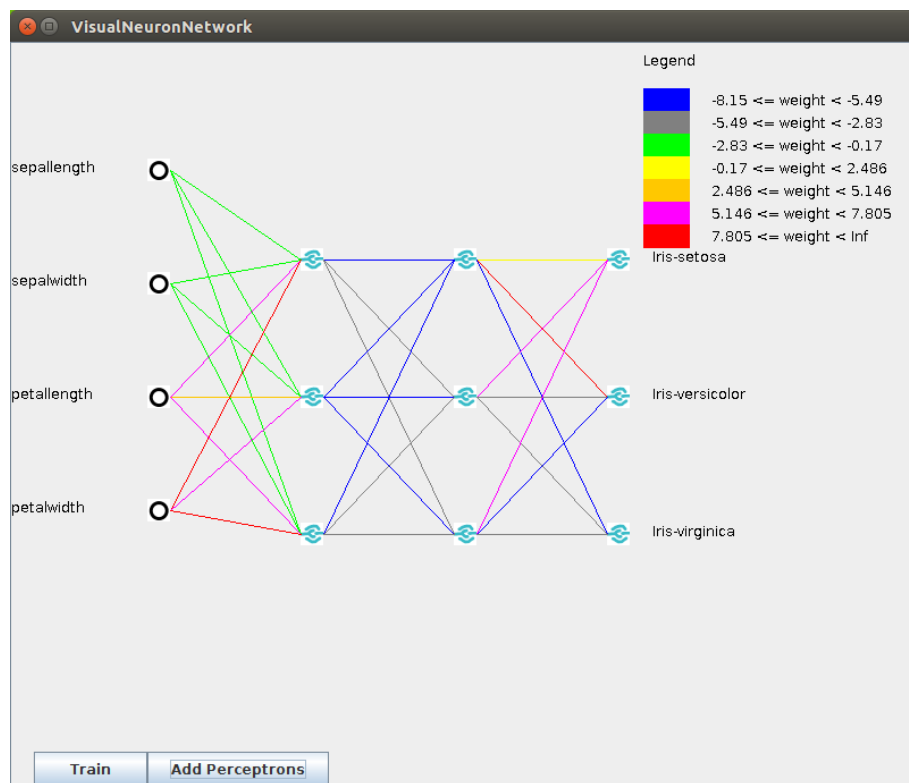
Implementation of the AQ15 classification algorithm described in [13]. The classifier uses a set of rules generated by a covering algorithm to classify data objects.

Parameters:

- *coverage* - as in Section 9.2.2

- *searchWidth* - as in Section 9.2.2
- *margin* - as in Section 9.2.2
- *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

11.7 Neural network



Authors: Jakub Sakowicz, Damian Wójcik

Rseslib class path:

`rseslib.processing.classification.neural.NeuronNetwork`

Description:

The classifier runs a number of rounds, in each round it shuffles randomly the training set and trains a network starting with the same network structure but with random connection weights. The network with the best accuracy on a

validation set is selected as the final model. The number of rounds is determined by the time limit given as a parameter.

In each round the classifier updates the connection weights using the classical backpropagation algorithm and sigmoid activation functions for all neurons [21]. A round ends when the network does not improve anymore or 75 iterations is run.

Neural networks can be visualized using *QMAK* program (see Chapter 13). Visualization of a model presents the neurons and the connections between them. The neurons from the last layer correspond to decisions. The color of a connection represents its weight. A user can select a neuron to display the exact weights of its input connections and its bias. Visualization can also display the learning process (after setting the *showTraining* parameter). A user can modify the network by adding new neurons in hidden layers and retraining the network.

Visualization of single object classification presents also the strength of the output signal from each neuron with intensity of its color and the exact value of the signal when a node is selected.

Parameters:

- *timeLimit* - time limit on searching for the optimal network (in seconds)
- *automaticNetworkStructure* (TRUE/FALSE) - if TRUE, the classifier uses a network structure with one hidden layer and the number of neurons in this layer computed algorithmically; if FALSE, the network structure is defined by a user
- *hiddenLayersSize* - used only if *automaticNetworkStructure* is FALSE, it defines the numbers of neurons in the hidden layers (separated by semicolons), for example the value 7;5;3 means that the network structure has 3 hidden layer, the first hidden layer has 7 neurons, the second layer has 5 neurons and the third layer has 3 neurons
- *initialAlpha* - initial value of the learning speed coefficient α in the back-propagation algorithm, the coefficient decreases over time
- *targetAccuracy* - target accuracy of classification (%); when the target is achieved on the validation set the learning process is stopped
- *showTraining* (TRUE/FALSE) - can be set to TRUE only in *QMAK*, then the network displays the changing connection weights during training

11.8 Naive Bayes

Author: Łukasz Ligowski

Rseslib class path:

`rreslib.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:

$$P(dec = d | \vec{x}) = \frac{P(\vec{x} | dec = d) \cdot P(dec = d)}{P(\vec{x})}$$

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*:

$$p(x | dec = d) = \frac{N_x + \frac{m}{Q}}{N + m}$$

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:

$$p(x | dec = d) = \frac{1}{Nh} \sum_i^N \phi\left(\frac{x - x^i}{h}\right)$$

where N is the number of objects in a decision class, h is smoothness parameter (the greater h value the smoother distribution), x_i is a value of a training object from a given decision class on an estimated attribute and ϕ is a kernel function. There are two types of such functions:

$$\begin{aligned} \phi(y) &= \begin{cases} 1 & \text{if } |y| \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases} && \text{hypercube} \\ \phi(y) &= \frac{1}{\sqrt{2\pi}} \left(\exp - \frac{|y|^2}{2} \right) && \text{Gaussian} \end{aligned}$$

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

Parameters:

- *mEstimateParameter* - *m*-estimate parameter, used for symbolic values

- *kernel* - type of kernel function, used for numerical values:
 - *gaussian*
 - *hypercube*
- *smoothness* - the parameter smoothing kernel functions, defines h in the formula for function density

11.9 Support vector machine

Author: Witold Wojtyra

Rseslib class path:

`rseslib.processing.classification.svm.SVM`

Description:

Detailed description and analysis of SVM model and classifier can be found in [34]. The SVM classifier classifies only numeric data. Data objects are treated as vectors of \mathbb{R}^n space, where n is the number of attributes. Using kernel transformations the classifier projects data from \mathbb{R}^n to H , where searching for dependencies between data is simpler. Classifier training is based on finding hyper-plane in H that 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 $\frac{k(k-1)}{2}$ binary classifiers are constructed and the decision for each data object is selected by voting (the winner is the most voted decision). To make the method insensitive to noise in data, it is possible to classify with error (then some training objects are misclassified - we aim at obtaining the most general classification model).

The output of the training phase are weight coefficients α for each training data object. The coefficients have non-zero values for support vectors (the name of the method is taken from these vectors). The classification phase calculates a decision function depending on α parameters.

Selection of kernel transformation has a large influence on classification. Kernel transformations enable to find complex dependencies in data. The following kernel transformations are implemented (x and y denote 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)

$$K(x, y) = e^{-\frac{\|x - y\|^2}{2\sigma^2}}$$

$\sigma \in \mathbb{R}$ -standard deviation

4. Exponential transformation

$$K(x, y) = e^{-\frac{\|x - y\|}{2\sigma^2}}$$

$\sigma \in \mathbb{R}$

5. Sigmoid transformation

$$K(x, y) = \tanh(\rho \langle x \cdot y \rangle + \theta)$$

$\rho, \theta \in \mathbb{R}$

Parameters:

- *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 compared. It is used during heuristic selection of points that are optimized. Too great value of this parameter causes that too many α coefficients are optimized (the number of iterations increases) and it can result in noticeable decrease of method efficiency. Too small values of this parameter make the method sensitive to rounding errors
- *epsilon* – the classifier stops where all input vectors satisfy Karush-Kuhn-Tucker conditions. The epsilon coefficient denotes acceptable error during calculation of condition value for every point. The greater coefficient, the faster the method but the smaller classification accuracy
- *kernel* – kernel transformation, the possible values are: *linear*, *polynomial*, *rbf*, *exponential*, *sigmoid*
- *polynomial_degree* – polynomial transformation parameter
- *polynomial_add* – polynomial transformation parameter

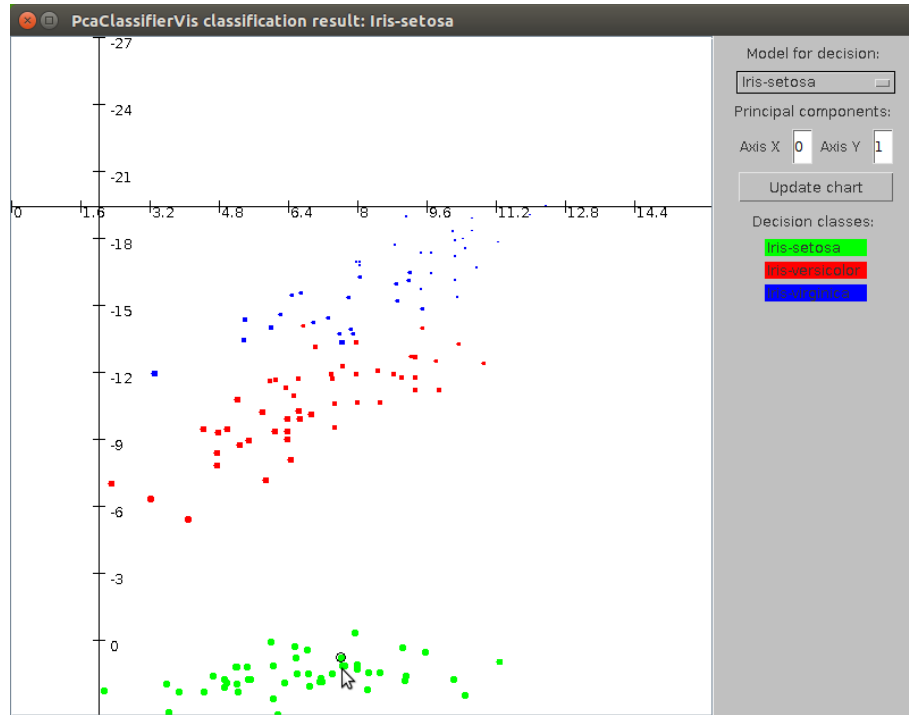
- *rbf_sigma* – gaussian transformation parameter
- *expotential_sigma* - exponential transformation parameter
- *sigmoid_kappa* - sigmoid transformation parameter
- *sigmoid_theta* - sigmoid transformation parameter

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** achives the best accuracy for C parameter from $\langle 0.05, 0.5 \rangle$.

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.

11.10 Classifier based on principal components analysis



Authors: Rafał Falkowski, Łukasz Kowalski

Rseslib class path:

`rseslib.processing.classification.parameterised.pca.PcaClassifier`

Description:

The classifier finds a separate model of principal components for each decision class using Oja-RLS rule. A detailed description and analysis of this classification model can be found in [3, 4].

The classifier can be visualized using *QMAK* program (see Chapter 13). Visualization of PCA classifier projects all training objects on the plane spanned by a pair of principal components of the model for one of decision classes. Different colors are used for objects with different decisions. The objects close to the plane are represented with bigger dots, the more distant the objects are from the plane the smaller dots represent them. Selection of the model and the principal components used to project objects can be changed by a user.

Visualization of single object classification by a PCA classifier marks additionally the projection of the classified object with a black circle.

Parameters:

- *principalSubspaceDim* - maximal number of principal components for a single decision class; the actual number of principal components is optimized during training

11.11 Classifier based on local principal components analysis

Author: Rafał Falkowski

Rseslib class path:

`rseslib.processing.classification.parameterised.pca.LocalPcaClassifier`

Description:

A detailed description and analysis of the classification model are available in [4]. The classifier finds several local models for each decision class.

Parameters:

- *principalSubspaceDim* - like in Section 11.10
- *noOfLocalLinearModels* - number of local models created for every decision class

11.12 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:

- *baggingWeakClassifiersClass* - classifier type used as “weak” classifier, given as class path
- *baggingNumberOfIterations* - number of iterations training “weak” classifiers
- *baggingUseWeakClassifiersDefaultProperties* (TRUE/FALSE) - if TRUE the classifier uses default parameters of “weak” classifier, if FALSE Bagging classifier expects that besides its parameters it is provided with all parameters of weak classifier as well.

11.13 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 [22, 23]. The experimental results from enhancing rule classifiers from Rseslib with AdaBoost method can be found in [29].

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 12

WEKA

Four Rseslib classifiers with full configuration:

- Rough set based rule classifier (Section 11.1)
- K nearest neighbours / RIONA (Section 11.2)
- K nearest neighbours with local metric induction (Section 11.3)
- RIONIDA (Section 11.4)

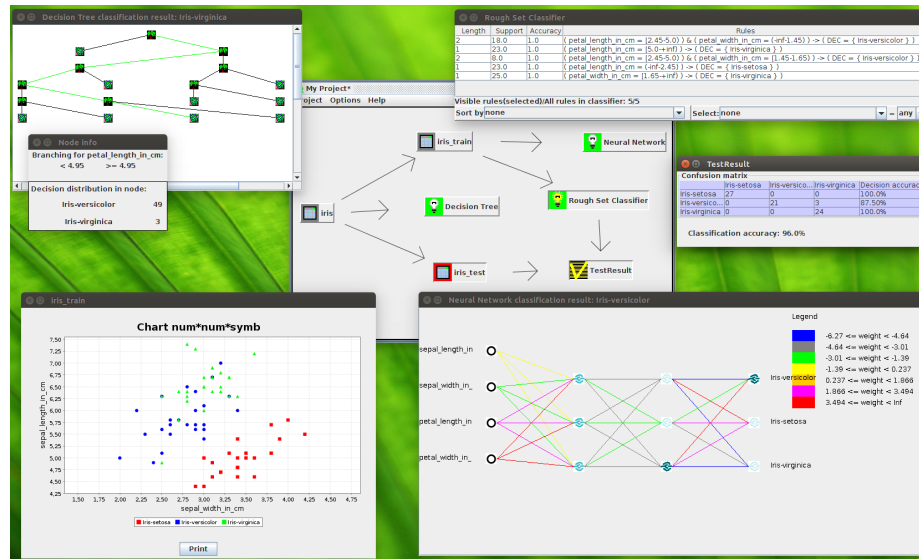
are available as an official Weka package for testing and experimenting in Weka. The package requires Weka version 3.8.0 or newer. To install Rseslib classifiers in Weka use the following steps:

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

Setting the option `debug` to `True` make Rseslib classifiers report progress of learning to the console.

Chapter 13

QMAK: Interaction with classifiers and their visualization



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

QMAK [32] is a graphical user interface dedicated to Rseslib library. Interaction with trained classifiers and visualization of classification models and classification process are the main features of the tool. It helps to understand why a

trained model selects a particular decision and how much it is sure of the assigned decision. The tool allows a user to find out which element of a model needs to be improved and to modify a model interactively. QMAK provides the following features:

- visualization of data, classifiers and single object classification
- interactive classifier modification by a user
- classification of test data with presentation of misclassified objects
- experiments comparing classification accuracy of classifiers with various test types: single train-and-classify test, cross-validation, multiple test with random train-and-classify split, multiple cross-validation

Five Rseslib classifiers provides model visualization and classification visualization in QMAK:

- Rough set based rule classifier (Section 11.1)
- K nearest neighbors (Section 11.2)
- C4.5 decision tree (Section 11.5)
- Neural network (Section 11.7)
- Classifier based on principal component analysis (Section 11.10)

The details of visualization of a particular classifier can be found in the specified section.

Users can implement new classifiers and their visualization (see Section 10.4) and add them easily to QMAK. It does not require any change in QMAK itself. A new classifier can be added using menu or in the configuration file.

To run QMAK download the latest *rseslib* package from <http://rseslib.mimuw.edu.pl>, unpack the package and run the script *qmak.sh* (on Linux) or *qmak.bat* (on Windows). If Rseslib is compiled from the source then to start QMAK run the class `rseslib.qmak.QmakMain` with Weka jar, *jfreechart-0.9.21.jar* and *jcommon-0.9.6.jar* added to the class path.

5-minute video demonstrating QMAK is available at <http://rseslib.mimuw.edu.pl/qmak>. Help on QMAK can be found in the main menu of the application. More detailed description of QMAK can be found in [32].

Chapter 14

SGM: Computing many experiments on many computers/cores

Author: Rafał Latkowski

Simple Grid Manager (SGM) is a tool for running many Rseslib-based experiments on a cluster of interconnected computers. SGM simplify running massive experiments (many experiments defined in script files) so it can be used to automate experiments even without utilizing parallel/distributed functionality (i.e. server & node on single machine). Additionally it allows to utilize many cores on single machine (i.e. many nodes on single machine). User do not need to explicitly configure cluster. Each computer successfully connected with server (or node relying mechanism) compose the cluster. Moreover user can write own classes similar to the other in `rseslib.processing.classification` and use them in massive/distributed experiments.

The main features of the program:

- Capability of *train-and-test* experiments applied to any classifier from *Rseslib* library (or other similar classes)
- Optimized to efficiently work in networks with unreliable connections:
 - UDP instead of TCP communication to minimize delays & unreliable network issues.
 - Lack of calls that can block network subsystem of operating system (e.g. reverse-DNS-lookup).
 - Communication between SGM components designed to work with existing firewalls and switches (e.g. to utilize computers in company network).

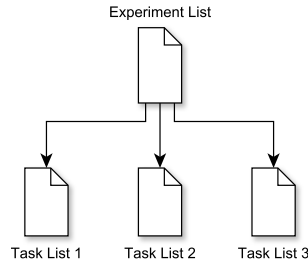


Figure 14.1: Hierarchical structure of experiment definition

- Relying mechanism between nodes (DAG architecture) to bypass NAT or firewalls on knot hosts.

- Multiplatform (java)

To run distributed experiment do the following steps:

1. Download the latest *rseplib* package from <http://rseplib.mimuw.edu.pl>
2. Copy the package to each machine and unpack it. (In modern version of SGM (3.x) we removed the functionality of transferring missing data files due to efficiency/reliability)
3. Prepare the experiment definition file (as described later) on the machine with the server
4. On one machine start the server (*sgm-server.sh* script on Linux, *sgm-server.bat* script on Windows) passing the experiment definition file as the parameter, e.g. on Linux:
`./sgm-server.sh <experiments file>`
5. On each machine start the client (*sgm-client.sh* script on Linux, *sgm-client.bat* script on Windows) providing the server name or IP address as the parameter, e.g. on Linux:
`./sgm-client.sh <server name/address> [options]`

14.1 Experiment definition & running SGM Server

Experiments are defined in hierarchical way (see Figure 14.1). In order to successfully run SGM Manager user need to define her/his experiment in two types of files: the experiment file and the task file(s). Each task file contains the list of tasks to be executed. The server outputs one file with results for each task 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 file with tasks is:

```

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

```

In the standard installation there is predefined example script “experiment.txt” for experiment file and “tasks.txt” for the task file. The following example of a single task definition in one line of task file:

```

rseslib.processing.classification.rules.roughset.RoughSetRuleClassifier
data/att_1.trn data/att_1.tst
Discretization=EqualWidth;Reducts=AllLocal;
IndiscernibilityForMissing=DiscernFromValue;
DiscernibilityMethod=OrdinaryDecisionAndInconsistenciesOmitted;
MissingValueDescriptorsInRules=TRUE

```

The definition of each job must be contained in a single line of the task file. Usually user does not need to specify each parameter of a classifier, but it is better to verify it first with current version of Rseslib. If a parameter is not specified then the default value is used. We can use any class that have constructor with 3 parameters of the classes: Properties, DoubleDataTable, Progress (as all of the classifiers in rseslib.processing.classification) and that implements interface rseslib.processing.classification.Classifier. Any user can write its own class, make available on class path in java runtime parameters and execute her/his own experiments based on newly developed class.

The format of the output file is similar to the task file. At the end of task description each line is appended with the task results:

```

stat1=val1;stat2=val2;stat3=val3;...

```

As described in previous section once we prepared experiment file and one or many task files we are ready to start server using example scripts (sgm-server.[sh|bat]) with mandatory parameter of experiment file name or by invoking class simplegrid.BatchManagerMain with mandatory parameter of experiment file name. Server displays in console messages related to initializing 16 threads for UDP communication on predefined port numbers and with each

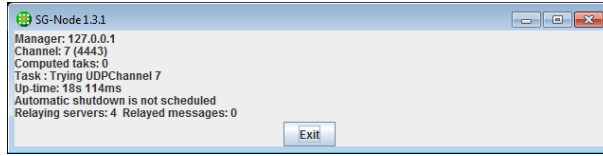


Figure 14.2: SGM-Node GUI displays current status of node and allows terminating node manually.

communication with a node it displays a message related to sent task or received result to/from a node. At each message displayed server also displays statistics: number of nodes active (contacted) in last 1-5-60 minutes as well as size of the queue of not finished tasks already read from task file (Queue Size) and number of tasks currently distributed for execution at nodes (In Execution).

At the end of experiments, when number of nodes exceeds number of experiments still to be executed, server tries to balance the load between machines by redistributing last jobs to all free nodes (race on last tasks). When we are sure that all tasks has been computed (e.g. by faster nodes) we can kill server manually. In other case server will wait until last task is finished and then shutdown slowly one thread at once on configured network timeout (currently it takes 16×5 seconds).

14.2 Running SGM-Node

SGM Node has its graphical user interface (see Figure 14.2) that displays current status of connection and computations as well as allows user terminating node manually. We can run node many times on the same machine (see next section on cluster architecture). The node can be started either by running a script `sgm-client.[sh|bat]` with mandatory parameter of server name/address or directly by invoking class `simplegrid.NodeMain` with mandatory parameter of server name/address. The node can connect with server with some delay due to retrying communication on different ports with default timeout set currently to 5 seconds.

Node can take additional parameters:

- `-DIE <time>` (e.g. `-DIE 15:49` or `-DIE "SU 23:00"`) — Sets scheduled shutdown to particular hour:minute or day-of-week (2 characters) and hour:minute. The node will terminate after selected amount of time. This functionality is useful if you wish to setup experiments e.g. over the weekend, when all computers are not used and terminate automatically on Monday morning before employees come back to work.
- `-FILEDEBUG` — In many cases experiments terminates due to bad parameters, lack of data files or insufficient memory. In order to analyze

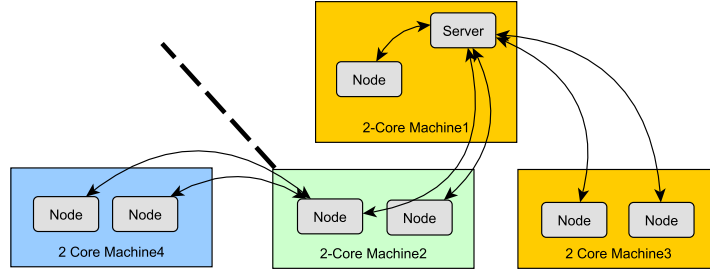


Figure 14.3: Example cluster architecture in case if Machine4 has no direct connection to server located on Machine1 (e.g. due to NAT), but can be reached by relying messages with SGM nodes on Machine2.

reason of error we can set the option `-FILEDEBUG` that writes all error messages related to experiment execution and communication with server to a file named `"sgm_node_log.txt"` in current directory.

- `-RELAY <list of relying nodes>` — Enables relying connection between nodes to avoid NAT and firewalls. Referring to an example from next section (see Figure 14.3) we need to include this option on Machine4 from that example and put Machine2 address on the list of relying nodes. List can be separated by comma or semicolon.
- `-FORCERELAY` — Even when relying option is configured, node still tries to communicate with server directly and only if it fails then it tries to use relying nodes. If we know that the direct communication is permanently not possible and re-trying connection with server is only waste of time (due to timeout-wait on each trial) we can put option `-FORCERELAY`. With this option node when successfully connects to a relay node then it never tries again direct communication with the server.

Moreover we can use standard java run-time environment options (e.g. `-Xmx` to set more working memory). In case of fast infinite loops of scheduling tasks user should verify `sgm_node_log.txt` (option `-FILEDEBUG`) to analyse error. It usually is caused by standard error on the node side (e.g. insufficient memory). All nodes also need to have data files available in their filesystems in the same directory structure as specified in task file. It is due to the fact that in modern version of Simple Grid ($>1.x$) there is no functionality of data file transfer.

14.3 Cluster architecture and message relying

Cluster of nodes in SGM is basically a tree having the server as a tree root and SGM nodes as leaves or internal tree nodes (in case of message relying). To

utilize existing cores on available machines we can start more than one node on single machine. For example on Figure 14.3 we have example how to utilize 2-core machines.

In distributed networks we sometimes face issues related to NAT (Network Address Translation) or other type of firewalling/traffic filtering that disallows direct connection between some machines (e.g. Machine4 has no direct connectivity to Machine1 on Figure 14.3). If any single machine is available for indirect connection (e.g. Machine2 on Figure 14.3) we can use mechanism of message relaying. Each SGM node can relay messages (UDP datagrams) from its child (nodes on Machine4) to its parent (server on Machine1).

Chapter 15

Command line programs

15.1 Compute and write reducts

An exemplary program computing reducts (see Chapter 8) and writing them to a file is implemented by the class `rseslib.example.ComputeReducts`. The program accepts the following arguments:

```
java -cp ... rseslib.example.ComputeReducts
      [-d <discretization>] [-dcfg <discr config file>]
      [-r <reducts>] [-rcfg <reducts config file>]
      <data file> [<output file>]
```

Warning! To use data in ARFF format the Weka jar must be added to the class path while running the program (see Section 4.1.1).

The argument `<discretization>` selects a type of discretization applied to data before an algorithm computing reducts is run. The possible values are (see Section 6.2): *None*, *EqualWidth*, *EqualFrequency*, *OneRule*, *EntropyMinimizationStatic*, *EntropyMinimizationDynamic*, *ChiMerge*, *MaximalDiscernibilityHeuristicGlobal*, *MaximalDiscernibilityHeuristicLocal*. The argument is optional, the default value is *MaximalDiscernibilityHeuristicLocal*.

The optional argument `<discr config file>` enables to provide the path to a file with customized configuration of the selected type of discretization. The template configuration file with default parameter values for each configurable type of discretization can be found in the resource directory in Rseslib sources: `src/main/resources/rseslib/processing/discretization/`.

The argument `<reducts>` selects an algorithm computing reducts. The possible values are (see Section 8.3): *AllLocal*, *AllGlobal*, *OneJohnson*, *AllJohnson*,

PartialLocal, *PartialGlobal*. The argument is optional, the default value is *AllGlobal*.

The optional argument `<reducts config file>` enables to provide the path to a file with customized configuration of the selected algorithm computing reducts. The template configuration file with default parameter values for each algorithm computing reducts can be found in the resource directory in Rseslib sources: *src/main/resources/rseslib/processing/reducts/*.

The argument `<data file>` is the file with data used to compute reducts.

The argument `<output file>` is the file the program writes reducts to. If not given the program writes reducts to the file *reducts.txt*.

15.2 Compute and write rules

An exemplary program computing rules (see Chapter 9) and writing them to a file is implemented by the class `rseslib.example.ComputeRules`. The program accepts the following arguments:

```
java -cp ... rseslib.example.ComputeRules
      [-d <discretization>] [-dcfg <discr config file>]
      [-r <rules>] [-rcfg <rules config file>]
      <data file> [<output file>]
```

Warning! To use data in ARFF format the Weka jar must be added to the class path while running the program (see Section 4.1.1).

The argument `<discretization>` selects a type of discretization applied to data before an algorithm computing rules is run. The possible values are (see Section 6.2): *None*, *EqualWidth*, *EqualFrequency*, *OneRule*, *EntropyMinimization-Static*, *EntropyMinimizationDynamic*, *ChiMerge*, *MaximalDiscernibilityHeuristicGlobal*, *MaximalDiscernibilityHeuristicLocal*. The argument is optional, the default value is *MaximalDiscernibilityHeuristicLocal*.

The optional argument `<discr config file>` enables to provide the path to a file with customized configuration of the selected type of discretization. The template configuration file with default parameter values for each configurable type of discretization can be found in the resource directory in Rseslib sources: *src/main/resources/rseslib/processing/discretization/*.

The argument `<rules>` selects an algorithm computing rules, either from reducts or AQ15 or accurate rules. The possible values are (see Section 9.2): *AllLocal*, *AllGlobal*, *OneJohnson*, *AllJohnson*, *PartialLocal*, *PartialGlobal*, *AQ15*, *Accurate*. The argument is optional, the default value is *AllGlobal*.

The optional argument `<rules config file>` enables to provide the path to a file with customized configuration of the selected algorithm computing rules.

The template configuration file with default parameter values for each algorithm computing rules can be found in the resource directory in Rseslib sources: *src/main/resources/rseslib/processing/rules/*.

The argument `<data file>` is the file with data used to compute rules.

The argument `<output file>` is the file the program writes rules to. If not given the program writes rules to the file *rules.txt*.

15.3 Cross-validation on Rseslib classifiers

An exemplary program executing cross-validation test (see Section 10.7) for all Rseslib classifiers is implemented by the class `rseslib.example.CrossValidation`. As arguments a user provides the number of cross-validation folds and a data file to be tested (with a header included), e.g.:

```
java -cp ... rseslib.example.CrossValidation 5 data/heart.dat
```

Warning! To use data in ARFF format the Weka jar must be added to the class path while running the program (see Section 4.1.1).

A header must be included in the input files.

The program writes data statistics, progress information and classification results to the standard output. The program contains an example how to test a classifier with non-default parameters.

15.4 Train and test Rseslib classifiers

An exemplary program executing a single test (see Section 10.6) for all Rseslib classifiers is implemented by the class `rseslib.example.TrainAndTest`. As arguments a user provides either one data file or two data files (the training and the test tables), e.g.:

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

Warning! To use data in ARFF format the Weka jar must be added to the class path while running the program (see Section 4.1.1).

A header must be included in the input files. If only one data file is provided the program splits the data randomly into the training and the test parts with the ratio 2:1.

The program writes data statistics, progress information and classification results to the standard output. The program contains an example how to test a classifier with non-default parameters.

Bibliography

- [1] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.
- [2] F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer Academic Publishers, Dordrecht, 1990.
- [3] K. I. Diamantaras and S. Y. Kung. *Principal Component Neural Networks: Theory and Applications*. John Wiley & Sons, Inc., New York, 1996.
- [4] R. Falkowski. Metoda składowych głównych w rozpoznawaniu obiektów. Master’s thesis, Faculty of Mathematics, Informatics and Mechanics, Warsaw University, 2004.
- [5] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1027. Morgan Kaufmann, 1993.
- [6] G. Góra. *Combining instance-based learning and rule-based methods for imbalanced data*. PhD thesis, Warsaw University, 2021.
- [7] G. Góra and A. G. Wojna. RIONA: a classifier combining rule induction and k-nn method with automated selection of optimal neighbourhood. In *Proceedings of the 13th European Conference on Machine Learning*, volume 2430 of *LNCS*, pages 111–123. Springer-Verlag, 2002.
- [8] R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine learning*, 11(1):63–90, 1993.
- [9] M. Jałmużna. Porównawcza analiza algorytmów dyskretyzacji. Master’s thesis, Faculty of Mathematics, Informatics and Mechanics, Warsaw University, 2009.
- [10] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.
- [11] R. Kerber. Chimerge: Discretization of numeric attributes. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 123–128. Aaai Press, 1992.

- [12] R. Latkowski. Flexible indiscernibility relations for missing attribute values. *Fundamenta Informaticae*, 67(1-3):131–147, 2005.
- [13] R. S. Michalski, I. Mozetic, J. Hong, and H. Lavrac. The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 1041–1045, 1986.
- [14] M. Moshkov, M. Piliszczyk, and B. Zielosko. Partial covers, reducts and decision rules in rough sets: Theory and applications. *Studies in Computational Intelligence*, 145, 2008.
- [15] H. S. Nguyen. *Discretization of Real Value Attributes: A Boolean Reasoning Approach*. PhD thesis, Warsaw University, 1997.
- [16] H. S. Nguyen and D. Ślęzak. Approximate reducts and association rules - correspondence and complexity results. In N. Zhong, A. Skowron, and S. Ohsuga, editors, *Proceedings of the International Workshop on Rough Sets, Fuzzy Sets, Data Mining, and Granular-Soft Computing*, volume 1711 of *LNCS*, pages 137–145. Springer, 1999.
- [17] W. Ogórek. Zbiory przybliżone w zadaniu generowania reguł decyzyjnych. Master’s thesis, Wrocław University of Science and Technology, 2011.
- [18] Z. Pawlak. *Rough Sets - Theoretical Aspects of Reasoning about Data*. Kluwer Academic Publishers, Dordrecht, 1991.
- [19] Z. Pawlak and A. Skowron. Rudiments of rough sets. *Information sciences*, 177(1):3–27, 2007.
- [20] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
- [21] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.
- [22] R. E. Schapire. A brief introduction to boosting. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1401–1406, 1999.
- [23] R. E. Schapire. Theoretical views of boosting. In *Proceedings of the Fourth European Conference on Computational Learning Theory*, pages 1–10. Springer, 1999.
- [24] A. Skowron. Boolean reasoning for decision rules generation. In J. Komorowski and Z. W. Raś, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, volume 689 of *LNCS*, pages 295–305. Springer, 1993.

- [25] A. Skowron and J. W. Grzymała-Busse. From rough set theory to evidence theory. In R. R. Yager, J. Kacprzyk, and M. Fedrizzi, editors, *Advances in the Dempster-Shafer Theory of Evidence*, pages 193–236. Wiley, New York, 1994.
- [26] A. Skowron and C. Rauszer. The discernibility matrices and functions in information systems. In R. Slowinski, editor, *Intelligent Decision Support, Handbook of Applications and Advances of the Rough Sets Theory*, pages 331–362. Kluwer Academic Publishers, Dordrecht, 1992.
- [27] A. Skowron and A. Wojna. K nearest neighbors classification with local induction of the simple value difference metric. In *Proceedings of the 4th International Conference on Rough Sets and Current Trends in Computing*, volume 3066 of *LNCS*, pages 229–234. Springer-Verlag, 2004.
- [28] R. Słowiński and J. Stefanowski. Rough classification in incomplete information systems. *Mathematical and Computer Modelling*, 12(10-11):1347–1357, 1989.
- [29] S. Stawicki. Wzmacnianie klasyfikatorów regułowych. Master’s thesis, Faculty of Mathematics, Informatics and Mechanics, Warsaw University, 2007.
- [30] A. Wojna. Center-based indexing for nearest neighbors search. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 681–684. IEEE Computer Society Press, 2003.
- [31] A. Wojna. Analogy-based reasoning in classifier construction (phd thesis). *LNCS Transactions on Rough Sets IV*, 3700:277–374, 2005.
- [32] A. Wojna, K. Jachim, Ł. Kosson, Ł. Kowalski, D. Mański, M. Mański, K. Mroczek, K. Niemkiewicz, R. Piszczatowski, M. Próchniak, T. Romańczuk, P. Skibiński, M. Staszczuk, M. Szostakiewicz, L. Tur, D. Wójcik, and M. Zuchniak. Qmak: Interacting with machine learning models and visualizing classification process. In *Proceedings of the 18th Conference on Computer Science and Intelligence Systems*, volume 35 of *ACSIS*, pages 315–318. IEEE, 2023.
- [33] A. Wojna and R. Latkowski. Rseslib 3: Library of rough set and machine learning methods with extensible architecture. *LNCS Transactions on Rough Sets XXI*, 10810:301–323, 2019.
- [34] W. Wojtyra. Implementacja klasyfikatora opartego o maszynowe wektory wspierających. Master’s thesis, Faculty of Mathematics, Informatics and Mechanics, Warsaw University, 2005.
- [35] J. Wróblewski. Covering with reducts - a fast algorithm for rule generation. In *Proceedings of the 1st International Conference on Rough Sets and Current Trends in Computing*, volume 1424 of *LNCS*, pages 402–407. Springer-Verlag, 1998.