KEEL Data-Mining Software Suite 3.0: Integration of New Algorithms

Isaak Triguero[a], Sergio González[b], Salvador García[b], Jesús Alcalá-Fdez[b], Julián Luengo[b], Alberto Fernández[b], María José del Jesus[c], José M. Moyano[d], Luciano Sánchez[1], Francisco Herrera[b]

[a]*School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham NG8 1BB, United Kingdom.*
[b]*Department of Computer Science and Artificial Intelligence, University of Granada, 18071, Granada, Spain.*
[c]*Department of Computer Science, University of Jaén, 23071, Jaén, Spain.*
[d]*Department of Computer Science and Numerical Analysis, University of Cordoba, 14071 Cordoba, Spain.*
[e]*Department of Computer Science, University of Oviedo, 33204, Gijón., Spain*

Contents

Email address: Isaac.Triguero@nottingham.ac.uk (Isaak Triguero)

1. Introduction to the KEEL codification features

This section is devoted to describing in detail how to implement or to import an algorithm into the KEEL software tool. The KEEL philosophy tries to include the fewest possible constraints for the developer, in order to ease the inclusion of new algorithms within this tool. In fact, each algorithm has its source code in a single folder and does not depends on a specific structure of classes, making the integration of new methods straightforward.

The list of details that must be taken into account before codifying a method for the KEEL software is listed below:

- The programming language used is Java.
- In KEEL, every method uses a single text configuration file to extract the values of the parameters which will be employed during its execution (Subsection 1.1).
- The input data-sets follow a specific format that extends the "arff" files by completing the header with more metadata information about the attributes of the problem. Next, the list of examples is included, which is given in rows with the attribute values separated by commas (Subsection 1.2).
- The output format consists of a header, which follows the same scheme as the input data, and two columns with the output values for each example separated by a whitespace. The first value corresponds to the expected output, and the second one to the predicted value. All methods must generate two output files: one for training and another one for test (Subsection 1.3).
- Once the source code has been developed, all files must be stored in the corresponding folder of the KEEL directory. The proper selection of this folder will depend on the algorithm's paradigm, as will be explained below. Additionally, the executable of the method must be build so it can be added to any new experiment. This step implies the modification of a single file that includes all the information for the building of the *.JAR* files of the different methods (Subsection 1.4).
- Next, all methods must be "registered" in KEEL so that they can be selected within the experimental frame (Subsection 1.5).
- Finally, including use case files will help other users to understand the characteristics of the algorithm, for example its main working procedure, or the number and type of parameters (Subsection 1.6).

1.1. Configuration files

As stated previously, all methods included in KEEL use a single text configuration file to specify the route of the input and output files. Additionally, it contains the values for all the parameters needed for the execution. In accordance with the former, each configuration file has the following structure:

- *algorithm*: Name of the method. It is only used for informative purposes.
- *inputData*: A list with the input data files of the method. Any method in KEEL employs two mandatory input data files:

1. The **training** file, containing the data set which should be employed in the train phase of the method.
2. The **test** file, containing the data set which should be employed in the test phase.

In addition, any method excepting the *preprocessing* methods and the *test* methods specify a third file, the **validation** file. The validation file is a copy of the original train data employed at the start of the experiment. It is often employed for comparison tasks between the initial data and training data when it has been preprocessed.

The files in the inputData list must be separated by one space, being each one surrounded by quotation marks ("). If a validation file is employed by the method, the files will appear in the following order: $inputdata =< trainingfile >< validationfile >< testfile >$

If not, the order employed will be: $inputdata =< trainingfile >< testfile >$

- *outputData*: A list with the output data files of the method. Regarding these, all methods must define at least two output files: A train output file and a test output file. In addition, it is possible to define additional output files in the configuration file, for example to store the learned model, and/or any other information of the learning process.
- *parameters*: A list of parameters of the method, containing the name of each parameter and its value (one line is employed for each one). If the method needs a seed to initialize a random number generator, it must be the first parameter described, employing "seed" as name of the parameter.

It is important to fully describe its structure because any KEEL method must be able to read it completely, in order to get the values of its parameters specified in each execution. Next we show a valid example of a Method Configuration file (data files lists are not fully shown):

```
algorithm = Genetic Algorithm
inputData = '../datasets/iris/iris.dat' ...
outputData = '../results/iris/result0.tra' ...

Seed = 12345678
Number of Generations = 1000
Crossover Probability = 0.9
Mutation Probability = 0.1
...
```

Whenever a new experiment is created within the experimental frame of KEEL, all parameter files are automatically generated by the KEEL GUI, allowing the user to select the values of the parameters of any execution of the method. In order to do so, every algorithm (e.g. a preprocess method, a test, classification approach, and so on) is assigned to a "method description file" which will describes its main characteristics, i.e. the configuration parameters. Specifically, this information is codified into an XML file.

KEEL **Method Description files** are located under the *../dist/algorithm* directory, inside of the folder

where its associated *.JAR* file is generated. There are 5 different types of algorithms to be considered, implying 5 folders where we can store this information:

- LQD: it stands for "Low Quality Data" algorithms. This is a special type of methods that works with uncomplete datasets.
- methods: most types of algorithms fall into this category, i.e. classification, regression, association rules, subgroup discovery, and so on.
- postprocess: here those methods that carry out the optimization for a learned model are stored. Typical examples are genetic fuzzy systems for tuning the knowledge base of fuzzy rule based systems.
- preprocess: any pre-processing approach must be placed here, i.e. instance selection, feature selection, or discretizers, among others.
- tests: it includes both statistical tests methods and visualization approaches.

Each method description file is usually named as the acronym of the associated algorithm. Additionally, in order to categorize this file a suffix is added to represent the family of the algorithm, as pointed out next:

- Instance Selection methods: *-TSS*
- Resampling methods: *-I*
- Feature Selection methods: *-FS*
- Discretization methods: *-D*
- Missing values methods: *-MV*
- Transformation methods: *-TR*
- Classification methods: *-C*
- Regression methods: *-R*
- Clustering methods: *-CL*
- Association rule mining methods: *-A*
- Subgroup discovery methods: *-SD*
- Imbalanced Classification methods: *-I*
- Post-processing methods: *-T*

As stated previously, each Method Description file is an XML composed by a unique root node, $< algorithmspecification >$. This node is divided into two parts, which are described below.

```
<algorithm_specification>
    Header
    Parameters
</algorithm_specification>
```

- **Header**: Basic information about the method. The header is composed by four nodes:
  - Name: The name of the method.
  - nParameters: The number of parameters of the methods (must be 0 or higher). Seed values

employed to initialize random number generators are not counted here.

- Seed: Defines if the method will need a seed to initialize a random number generator. Valid values are 1, if a seed is needed, or 0, if not.
- nOutput: The number of additional output files which will be generated by the method.

An example of the former is shown below:

```
<name>K Nearest Neighbors Classifier</name>
<nParameters>2</nParameters>
<seed>0</ seed>
<nOutput>1</ nOutput>
```

- **Parameters**: A list of parameters of the method.

  The parameters of the method are listed consecutively. A <parameter> node is employed to describe each one. Each <parameter> is composed by the following nodes:

  - Name: The name of the parameter.
  - Type: Type of parameter. KEEL defines four valid types:
    1. integer: An integer value. Can be positive, 0, or negative.
    2. real: A real value. The dot "." is employed as decimal separator.
    3. text: A string of text.
    4. list: A predefined list of text options

    When employing text parameters, no checking operations are done by the KEEL GUI. Thus, the use of list parameters is recommended when a fixed number of text options are defined, so the method does not have to check the parameters by itself.
  - Domain: The domain of the parameter. For list parameters is mandatory. For text parameters cannot be defined. For integer and real parameters is optional (if it is not defined, the KEEL GUI will not check its value).
    - lowerB: The lower value of the parameter (valid only in integer and real parameters).
    - upperB: The highest value of the parameter (valid only in integer and real parameters).
    - item: A text value for the parameter (it can be employed only in list parameters).
  - Default: Default value of the parameter.

  Regarding the former issues, a common parameters' list will have the following structure:

```
<parameter>
    <name>K Value</name>
    <type>integer</type>
    <domain>
        <lowerB>1</lowerB>
        <upperB>100</upperB>
    </ domain>
    <default>1</default>
</parameter>
```

1.2. Input Files

In KEEL, the data sets are managed by plain ASCII text files, with the *.dat* extension. Usually, they are located under the *../dist/data* directory, each one in its own folder (which also should contains the partitions created from the whole data set). In addition, preprocess methods will also create data files as its output, which will be placed on the *../datasets* directory of its experiment, using the name of the preprocessing approach as prefix.

This section describes the format employed to define them (which is fairly similar to WEKA arff format). Each KEEL data file is composed by two sections:

1. Header: Basic metadata describing the data set. Specifically, it is composed by the following information:

   - @relation: The name of the data set.
   - @attribute: Describes one attribute of the data (a column). It is possible to define three different types of attributes:

     (a) integer : **@attribute** $< name > integer[min, max]$

     (b) real : **@attribute** $< name > real[min, max]$

     (c) nominal : **@attribute** $< name > Value_1, value_2, \ldots, value_N$

     The $< name >$ is the identifier of the attribute. Its maximum length allowed is 12 characters. The min and max values for integer and real attributes, and the list of possible values for nominal attributes, are optional. If they are missing, the corresponding values will be extracted from the data by the KEEL data process module.
   - @inputs: Identifiers of the attributes which must be processed as inputs. @outputs : Identifiers of the attributes which must be processed as outputs.
     The @inputs and @outputs definitions are optional. If they are missing, all the attributes will be considered as input attributes, except the last, which will be considered as output attribute.

   Therefore, any header will have the following structure:

```
@relation bupa
@attribute mcv nominal a, b, c
@attribute alkphos integer [23, 138]
@attribute sgpt integer [4, 155]
@attribute sgot integer [5, 82]
@attribute gammagt integer [5, 297]
@attribute drinks real [0.0 , 20.0 ]
@attribute selector true , false
@inputs mcv, alkphos , sgpt , sgot , gammagt , drinks
@outputs selector
```

2. Data: Content of the dataset.

   The data instances are represented as rows of comma separated values, where each value corresponds to one attribute, in the order defined by the header. Missing or null values are defined as $< null >$ or ?. If the dataset corresponds to a classification problem, the output type must be nominal:

```
...
@attribute selector true, false
...
@outputs selector
@data
a, 92, 45, 27, 31, 0.0, true
a, 64, 59, 32, 23, <null>, false
b, 54, <null>, 16, 54, 0.0, false
...
```

   If the dataset corresponds to a regression problem, the output type must be real:

```
...
@attribute selector real [0.0, 20.0]
...
@outputs selector
@data
a, 92, 45, 27, 31, 0.0, 0.9
a, 64, 59, 32, 23, <null>, 17.5
b, 54, <null>, 16, 54 , 0.0, 3.5
...
```

1.3. Output Files

As stated at the beginning of this section, it is mandatory for every method in KEEL to produce at least two output files: A train results file (marked with the extension *.tra*) and a test results file (marked with the extension *.tst*). Although the method can employ additional output files to show more information about the process performed, those additional files must be handled entirely by the method. Thus, KEEL will only handle the two standards output files.

Both output files share the same structure: They are composite by the same header of the data employed as input of the method, and a set of rows (one for each instance of the data set) describing the expected outputs and the outputs obtained by the application of the method. Thus, they are structured as follows:

```
<Expected1,1> ... <Expected1,n><Method1,1> ... <Method1,n>
<Expected2,1> ... <Expected2,n><Method2,1> ... <Method2,n>
```

If it is desired to employ additional output files, they also can be created at the end of the execution on the method. These additional files will get its name from the configuration file, with the *.txt* extension. Also, it is important to recall that, in order to let the KEEL GUI automatically generate the names of these files, the number of additional outputs of the methods must be placed in the corresponding method description file (see Subsection 1.1).

1.4. Adding the source code and building the executable

The development of the method can be done in any programming environment. The only requirements are: The method must be developed with the Java programming language, and it must employ a package structure whose root will be the *keel/src/Algorithms* directory, where the sources of any KEEL method are located.

The question is where should be placed any new method within the tree directory of the KEEL source code. The answer depends on the typology of the algorithm that is to be implemented. There are 32 different folders, each one referring to a different kind of learning scheme, from "Associative Classification" to "Unsupervised Learning."

When the method was fully developed, and its relevant configuration files have been created, the last step is to add it to the *build.xml* file (an *ANT* script file [1]), so the new versions of KEEL could be able to build it inside the KEEL environment. The *build.xml* is a critical file, so one must be careful when modifying its content not to affect the remaining build tasks.

The *build.xml* changes dynamically with any new version of KEEL, thus its is not possible to fully describe its structure here. However, it is possible to describe which part of the file must be changed to allow the inclusion of new methods.

Firstly, the *jar* target must be found. It should have the following structure:

```
<target name=‘‘jar’’ depends=‘‘compile’’
description=‘‘Build jars’’>
```

The jar target is composed by a great number of tasks, every one dealing with the construction of a jar file for each method. Inside this target, the construction of the new jar file must be described as another task. Here is a valid example:

```
<jar
    jarfile =‘‘$distMet/KNN.jar’’ manifest=
    ‘‘$src/keel/Algorithms/Lazy_Learning/KNN/Manifest’’>
    <fileset dir=‘‘$bin’’ includes=
    ‘‘keel/Algorithms/Lazy_Learning/KNN/**/*.class’’/>
    <fileset dir=‘‘$bin’’ includes=
    ‘‘keel/Algorithms/Preprocess/Basic/**/*.class’’/>
    org/core/**/*.class
    keel/Dataset/**/*.class
    keel/Algorithms/Lazy_Learning/*.class’’/>
</jar>
```

The task must define the locations of the new jar file and their corresponding manifest file. Also, it must include the files from the classes which compose the method. Also, the files from the imported classes are required to fully describing the task.

[1] https://ant.apache.org/

If you are using any external library for your code, you must link it also in here, in order to reference the corresponding "jar files." If you are using an existing library from the "lib" folder, you must search its identifier within the *Libraries filesets* section at the beginning of the *build.xml* file.

For example, if you need the *jdom XML reader*, you will use the "lib_xml.jars" identifier within the *jar target* code as follows:

```
<jar
    jarfile="$src/keel/GraphInterKeel/resources/runkeel/runkeel.jar"
    manifest="$src/keel/RunKeelTxt/Manifest">
    <fileset dir="$bin"
        includes="keel/RunKeelTxt/**/*.class"/>
    <zipgroupfileset refid="lib_xml.jars"/>
</jar>
```

However, if you want to add a custom library to KEEL software, first of all you need to place the corresponding jar files into a new folder within the "lib" directory. In this way, it will be available from different source codes.

Afterwards, you should include the directives for linking the jar files of this library into the "classpath" of your algorithm. For this task, five steps are needed:

1. Add a new line to the *Properties of the project* to specify the current location of the new library.
2. Include all "jar" files into the *Libraries filesets*.
3. Add the former into the *Libraries path*.
4. Define a classpath for use throughout the buildfile.
5. Then, you can simply proceed as in the previous case.

The following example comprises these five steps:

```
<!-- Properties of the project-->
    <property name="src" location="src"/>
    ...
    <property name="lib_xml" location="lib/XML"/>
    <property name="lib_jama" location="lib/Jama"/>

...
<!-- Libraries filesets-->
...
    <fileset id="lib_xml.jars" dir="$lib_xml">
        <include name="**/*.jar"/>
    </fileset>

<fileset id="lib_jama.jars" dir="$lib_jama">
        <include name="**/*.jar"/>
```

```
    </fileset>

...
<!-- Libraries path-->
    <path id="lib.path">
        ...
        <fileset refid="lib_xml.jars"/>
<fileset refid="lib_jama.jars"/>
        ...
    </path>

...
<!-- Define a classpath for use throughout the buildfile -->
  <path id="GUI.classpath">
    <pathelement location="$src"/>
    <pathelement location="$bin"/>
    <pathelement location="$lib"/>
    ...
    <!-- include our own libraries -->
    <fileset refid="lib_xml.jars"/>
    ...
<fileset refid="lib_jama.jars"/>
    ...
   </path>

...
<jar
    jarfile="$distMet/IG-SGP.jar"
manifest="$src/keel/Algorithms/Instance_Generation/SGP/SGPAlgorithm.mf">
    <fileset dir="$bin" includes="keel/Algorithms/Instance_Generation/Basic/*.class" />
    ...
<zipgroupfileset refid="lib_jama.jars"/>
</jar>
```

1.5. Registering the method in KEEL

When the method have been fully coded, it must be registered in the KEEL configuration files, to allow the KEEL GUI to import the new method.

The first step is to create a **Method Description File** whose format was described previously in Section 1.1.

The second step involves modifying the **Master Description File** of each category method within the *dist/algorithm* folder. Currently, 18 categories are defined

1. Discretization
2. Educational Methods
3. Educational Preprocess

4. Feature Selection
5. Instance Selection
6. Methods
7. Methods Imbalanced
8. Methods Multi-instance
9. Methods Semi-Supervised Learning (SSL)
10. Postprocess
11. Preprocess
12. Preprocess Imbalanced
13. Subgroup Discovery
14. Tests
15. Tests Imbalanced
16. TransOthers
17. Visualize
18. Visualize Imbalanced

When the correct XML Master Description File have been found (please, ask to a KEEL project manager if it is not clear which file has to be modified), a new registry containing the definition of the method must be created. The KEEL master description file registers have the following structure:

```
<method>
    Header
    Input
    Output
</method>
```

These are described in detail next:

- Header: it is composed by four nodes:
    1. Name: The name of the method. It is used to identify the method. It is very important to use the same name given to the **Method Description File** to link both files.

2. Family: The category of the method. This implies the "folder" in which it will be placed within the *experimental frame* in the KEEL GUI. Depending on the type of the task to be carried out, the following "families" can be used:
   - Preprocessing:
     * Data Complexity
     * Discretization
     * Evolutionary Feature Selection
     * Evolutionary Training Set Selection
     * Feature Selection
     * Missing Values
     * Noisy Data Filtering
     * Training Set Selection
     * Transformation
   - Algorithms for classification/regression/unsupervised learning
     * Association Rules
     * Associative Classification
     * Clustering Algorithms
     * Crisp Rule Learning
     * Decision Trees
     * Evolutionary Crisp Rule Learning
     * Evolutionary Fuzzy Rule Learning
     * Evolutionary Fuzzy Symbolic Regresion
     * Evolutionary Neural Networks
     * Evolutionary Prototype Selection
     * Fuzzy Instance Based Learning
     * Fuzzy Rule Learning
     * Lazy Learning
     * Nested Generalized Learning
     * Neural Networks
     * Prototype Generation
     * Prototype Selection
     * Statistical Classifiers
     * Statistical Regression
     * Support Vector Machines

- Postprocessing: Fuzzy Rule Postprocessing
- Tests
  - Tests for classification
  - Tests for regression
- Visualization
  - Show results (classification)
  - Show results (regression)
  - Multiple results (classification)
  - Multiple results (regression)
  - Fingrams (fuzzy)

3. Jar File: The name of the Jar file which contains the method. It is also quite important to use the same name as the defined in the build file (Subsection 1.4).
4. Problem Type: The class of problems which can manage the method. There are 4 classes defined:
   (a) Classification, for supervised classification problems and subgroup discovery.
   (b) Regression, for regression problems.
   (c) Unsupervised, for unsupervised classification problems (e.g. clustering or association rule mining).
   (d) Unspecified, for any problem (supervised classification, unsupervised classification or regression). This is mainly used for preprocessing approaches that can be applied to any kind of Data Mining task.

An example for a discretization algorithm is shown next:

```
<name>Disc-UniformWidth</name>
<family>Discretizers</family>
<jar_file>Disc-UniformWidth.jar</jar_file>
<problem_type>unspecified</problem_type>
```

- Input and Output: The input and output parts defines the types of data which the method is able to manage, both in input data and output data. Their fields must specify which types are allowed, by employing "yes" and "no" keywords. A description of the fields is shown as follows:
  - Continuous: The method is able to work with continuous values.
  - Integer: The method is able to work with integer values.
  - Nominal: The method is able to work with nominal values.
  - Missing: The method is able to handle missing values.
  - Imprecise Value: The method is able to work with imprecise values.
  - Multiclass: The method is able to work with problem which defines more than 2 classes.

– Multioutput: The method is able to work with data which defines more than 1 output for each instance.

A simple example is shown below:

```
<continuous>Yes</continuous>
<integer>Yes</integer>
<nominal>Yes</nominal>
<missing>Yes</missing>
<imprecise>No</imprecise>
<multiclass>Yes</multiclass>
<multioutput>No</multioutput>
```

With this information, the KEEL GUI will raise an "alert" when the node for this algorithm is connected with an input that is not suitable for its execution, i.e. a dataset with real values as attributes.

When the header, input and output sections were completely defined, then the new registry can be placed inside the corresponding **Master Description File**. Below is shown a valid example of a complete registry:

```
<method>
    <name>Disc-UniformWidth</name>
    <family>Discretizers</family>
    <jarfile>Disc-UniformWidth.jar</jarfile>
    <problem type>unspecified</problem type>
    <input>
        <continuous>Yes</continuous>
        <integer>Yes</integer>
        <nominal>Yes</nominal>
        <missing>Yes</missing>
        <imprecise>No</imprecise>
        <multiclass>Yes</multiclass>
        <multioutput>No</multioutput>
    </input>
    <output>
        <continuous>No</continuous>
        <integer>No</integer>
        <nominal>Yes</nominal>
        <missing>Yes</missing>
        <imprecise>No</imprecise>
        <multiclass>Yes</multiclass>
        <multioutput>No</multioutput>
    </output>
</method>
```

Finally, we must point out that updating the **Master Description File** allows the method to be shown within the experimental frame of KEEL Software Suite.

1.6. Use cases

Although it is not mandatory according to KEEL encoding guidelines, whenever a new method is developed, it is important to document properly its functions and objectives. Also, users should be able to

look up relevant information about the method (a brief description, some references, the description of its parameters, etc.) when they select it for their experiments in KEEL.

To manage this information, the KEEL GUI defines the use case files, which are XML files containing all the relevant information needed to employ any KEEL method. These are located within the *../dist/help* directory. Each KEEL use case file is composed by 4 sections, as shown next:

```
<method>
    Name
    Reference
    General Description
    Example
</method>
```

We describe the content for each section below:

- Name: The name of the method. It is enclosed by $< name >$ tags:

  ```
  <name>Name of the method</name>
  ```

- Reference: A list of references associated with the method. They are enclosed each one by $< ref >$ tags.

  ```
  <reference>
      <ref>First reference</ref>
      <ref>Second reference</ref>
  </reference>
  ```

- General Description: It describes some common features about the method, as its objective, parameters, type of data which can be handle, etc. It is composed by the following fields:
  - Type: General type of method.
  - Objective: Objective of the method.
  - How work: A brief explanation about how the method works.
  - Parameter spec: A specification of each parameter of the method. They are enclosed each one by ¡param¿ tags.
  - Properties: Generic properties of the methods. Each field can contain "yes" or "no" strings, defining the following capabilities of the method:
    - Continuous: The method is able to work with continuous values.
    - Discretized: The method is able to work with discretized values.
    - Integer: The method is able to work with integer values.
    - Nominal: The method is able to work with nominal values.
    - Value Less: The method is able to handle missing values.
    - Imprecise Value: The method is able to work with imprecise values.

  This part of the XML will have the following structure:

```
<generalDescription>
    <type>General type of method .</type>
    <objective>Objective of the method .</objective>
    <howWork>Explanation of how it works .</howWork>
    <parameterSpec>
        <param>Parameter one</param>
        <param>Parameter two</param>
    </parameterSpec>
    <properties>
        <continuous>Yes</continuous>
        <discretized>Yes</discretized>
        <integer>Yes</integer>
        <nominal>Yes</nominal>
        <valueLess>Yes</valueLess>
        <impreciseValue>Yes</impreciseValue>
    </properties>
</generalDescription>
```

- Example: The last part of the use case is employed to show an example of utilization of the method. It can be comprised by any number of lines, though it is not recommended to place huge examples here.

  ```
  <example> A example of usage of the method. </example>
  ```

2. KEEL Source Code Template

Although the list of constraints for the implementation of a method within KEEL was shown to be short, our development team have created a simple template that manages those features regarding to the implementation of the method. Specifically, our KEEL template includes four classes:

1. Main: This class contains the main instructions for launching the algorithm. It reads the parameters from the file and builds the "algorithm object."

2. ParseParameters: This class manages all the parameters, from the input and output files, to every single parameter stored in the parameters file.

3. myDataset: This class is an interface between the classes of the API data-set and the algorithm. It contains the basic options related to data access.

4. Algorithm: This class is devoted to storing the main variables of the algorithm and to naming the different procedures for the learning stage. It also contains the functions for writing the obligatory output files.

The template can be downloaded following the link `http://www.keel.es/software/KEEL_template.zip`, which additionally supplies the user with the whole API data-set together with the classes for managing

files and the random number generator. Additionally, it includes two different coding examples, i.e. Naïve Bayes and the Chi et al.'s Fuzzy Rule Based Classification System.

Most of the functions of the classes presented above are self-explanatory and fully documented to help the developer understand their use. Nevertheless, throughout this section, we will describe the content of each one of these classes.

2.1. Main class

This class is devoted to initialize the program. In order to do so, it first reads the single parameter file included as argument (please refer to Section 1.1). Then, it gathers all the input parameters and builds the "Algorithm" object which will contain the main instructions for executing the method itself.

As it can be regarded below, the structure for this class is quite simple, just acting as the interface between the program call, and the main algorithm. Additionally, any interested user can add the computation of the elapsed time of the algorithm here.

```
public class Main {

    private parseParameters parameters;

    private void execute(String confFile) {
        parameters = new parseParameters();
        parameters.parseConfigurationFile(confFile);
        Algorithm method = new Algorithm(parameters);
        method.execute();
    }

     public static void main(String args[]) {
        Main program = new Main();
        System.out.println("Executing Algorithm.");
        program.execute(args[0]);
    }
}
```

2.2. Gathering the parameters: ParseParameters class

This class has made available in order to avoid users to cope with the reading of the parameters from the configuration file (Section 1.1). There is one public method to read all parameters from file, and additional public procedures for accessing and importing them to the program.

As it was shown in the previous section, in which the "Main" class was described, the first step is to call the "parseConfigurationFile" procedure that stores the links for the input and output files. It also reads line by line the value for each parameter.

Mandatory input (training, validation and test) and output files (training and test) are stored in different variables, and they can be accessed with an independent function (i.e. "getTrainingInputFile()"). The remaining auxiliary input and output files are gathered into a list of "String" values.

All parameters' values are also stored in a list of "Strings." This way, they need to be parsed in the main program in order to extract the integer or real value if it is the case. This will be explained later when referring to the "Algorithm" class. It is also important to know the actual position for each variable, as they will be referred by this index within the main program, using the "getParameter(int pos)" function.

Usually, this class does not need to be modified, as it fully supports the reading of the parameters of the program.

```
public class parseParameters {

  private String algorithmName;
  private String trainingFile, validationFile, testFile;
  private ArrayList <String> inputFiles;
  private String outputTrFile, outputTstFile;
  private ArrayList <String> outputFiles;
  private ArrayList <String> parameters;

  public parseParameters() {
      inputFiles = new ArrayList<String>();
      outputFiles = new ArrayList<String>();
      parameters = new ArrayList<String>();

  }

  public void parseConfigurationFile(String fileName) {
      StringTokenizer line;
      String file = Files.readFile(fileName);

      line = new StringTokenizer(file, "\n\r");
      readName(line);
      readInputFiles(line);
      readOutputFiles(line);
      readAllParameters(line);

  };

  ...

  public String getTrainingInputFile(){
      return this.trainingFile;
  };

  ...

  public String getParameter(int pos){
      return (String)parameters.get(pos);
  };

  ...

}
```

2.3. Managing the API Dataset: myDataset class

In order to ease the management for the input dataset files (see Section 1.2), the "myDataset" class includes the main procedures that serve as an interface between the API Dataset and the method to be included in KEEL Software Suited.

There are several public and private functions already implemented for this task that can be useful for the developer. It this section we enumerate some the main public functions:

- myDataset: is the builder of the method, and it just simply initialize the InstanceSet object of the API Dataset.
- readClassificationSet / readRegressionSet: this couple of functions are devoted to read the whole file and fulfill the variables that stores the examples of the input data. They also compute the statistics of the input attributes, such as the range, mean, standard deviation and number of instances per class (for classification problems), among others.
- getExample: it returns a single example of the dataset as an array of real values. Only input attributes are returned.
- getOutputAsString / getOutputAsInteger / getOutputAsReal: it returns the output value for a given example, or the whole list of outputs for all examples.
- getSize: it returns the number of examples of the problem.

We must point out that these and other functions are documented with detail so that any interested researcher can make use of them without problems. Additionally, in case of needing additional procedures for a given algorithm, these can be easily inserted within this class.

```
public class myDataset {

  private double[][] X;
  private double[] outputReal;
  private String[] output;

  private int nData;
  private int nVars;
  private int nInputs;

  private InstanceSet IS;

  public myDataset() {
      IS = new InstanceSet();
  }

  public double[] getExample(int pos) {
      return X[pos];
```

```
    }

    public void readClassificationSet(String datasetFile,
            boolean train) throws IOException {
        try {
            IS.readSet(datasetFile, train);
            nData = IS.getNumInstances();
            nInputs = Attributes.getInputNumAttributes();
            nVars = nInputs + Attributes.getOutputNumAttributes();

    ...

    }
}
```

2.4. Building the method itself: Algorithm class

The main effort of the implementation for any method is located in the "Algorithm" class. Here, we should call those procedures that carry out the learning process and we should be able to read the final model in order to give an output.

Fortunately, the developer only need to focus on the inner features of his/her method, as all the inner functions of the KEEL Software Suite are already included in this "Algorithm" class. In what follows, we describe the three main procedures that aid with the implementation task:

1. Algorithm: it is the builder of the class. First of all, it initializes the training, validation and test input data, and read its content from file (by means of the "myDataset" class). Then, the links for mandatory training and test output files are read. Finally, it reads and assigns all the parameters of the algorithm.

2. Execute: this is the core of the algorithm. The first step is to check whether a problem has raised during the initialization of the program, i.e. within the builder. Otherwise, we proceed with the algorithm's operations, which must be codified by the user. The last stage is filling the training and test output files according to the learned model.

3. doOutput: it is the procedure that computes the output for each example, as stated in Section 1.3. Depending whether we address a classification or regression problem, a different procedure is called in which the developer must state how the model provides the actual output.

```
public class Algorithm {

  myDataset train, val, test;
  String outputTr, outputTst;
  private boolean somethingWrong = false;
```

```
    public Algorithm(parseParameters parameters) {

        train = new myDataset();
        val = new myDataset();
        test = new myDataset();
        try {
            System.out.println("\nReading the training set: " +
                               parameters.getTrainingInputFile());
            train.readClassificationSet(parameters.getTrainingInputFile(),
                                true);
            System.out.println("\nReading the validation set: " +
                               parameters.getValidationInputFile());
            val.readClassificationSet(parameters.getValidationInputFile(),
                                false);
            System.out.println("\nReading the test set: " +
                               parameters.getTestInputFile());
            test.readClassificationSet(parameters.getTestInputFile(),
                                false);
        } catch (IOException e) {
            System.err.println("There was a problem while reading
                                the input data-sets: " + e);
            somethingWrong = true;
        }

        outputTr = parameters.getTrainingOutputFile();

    ...
    }
}
```

3. Encoding example using the "Steady-State Genetic Algorithm for Extracting Fuzzy Classification Rules From Data" method

Developing new algorithms in the KEEL software suite is very simple using the source code template presented in the previous section. Additionally, the effort needed for including and registering it into KEEL is also minimal according to the guidelines given in the first section. In order to unify both procedures, we will illustrate how to include a classical and simple method, the "Steady-State Genetic Algorithm for Extracting Fuzzy Classification Rules From Data" (SGERD) procedure. The details regarding the source code implementation will be given in Section 3.1. Then, the integration of the method itself will be explained in Section 3.2.

Finally, we should point out that the complete source code for the SGERD method (together with the needed classes for the fuzzy rule generation step) can be downloaded at `http://www.keel.es/software/SGERD_source.zip`.

3.1. Implementation of the SGERD algorithm

In this section, we will show how the provided template enables the programming within KEEL to be straightforward, since the user does not need to pay attention to the specific KEEL constraints because they are completely covered by the functions implemented in this template.

In particular, and as pointed out in Section 2, neither the Main nor the ParseParameters classes need to be modified, and we just need to focus our attention on the Algorithm class and the inclusion of two new functions in myDataset. We enumerate below the steps for adapting this class to this specific algorithm:

1. First of all, we must store all the parameters values within the constructor of the algorithm. Each parameter is selected with the `getParameter` function using its corresponding position in the parameter file, whereas the optional output files are obtained using the function `getOutputFile`. Furthermore, the constructor must check the capabilities of the algorithm, related to the data-set features, that is, whether it has missing values, real or nominal attributes, and so on. These operations are shown in the following source code, in which the operations that are added to the template are stressed in **boldface**:

```
public SGERD(parseParameters parameters) {

  train = new myDataset();
  val = new myDataset();
  test = new myDataset();
  try {
    System.out.println("\nReading the training set: " +
      parameters.getTrainingInputFile());
    train.readClassificationSet(parameters.getTrainingInputFile(),
              true);
    train.computeOverlapping();
    System.out.println("\nReading the validation set: " +
      parameters.getValidationInputFile());
    val.readClassificationSet(parameters.getValidationInputFile(),
              false);
    System.out.println("\nReading the test set: " +
      parameters.getTestInputFile());
    test.readClassificationSet(parameters.getTestInputFile(),
              false);
  }
  catch (IOException e) {
    System.err.println("There was a problem while reading the input
              data-sets: " + e);
    somethingWrong = true;
  }

  somethingWrong = somethingWrong || train.hasMissingAttributes();

  outputTr = parameters.getTrainingOutputFile();
```

```
    outputTst = parameters.getTestOutputFile();

    fileDB = parameters.getOutputFile(0);
    fileRB = parameters.getOutputFile(1);

    long seed = Long.parseLong(parameters.getParameter(0));

    Q = Integer.parseInt(parameters.getParameter(1));
    if ((Q < 1) || (Q > (14*train.getnInputs())))
        Q = Math.min((14*train.getnInputs()) / (2*train.getnClasses()), 20);

    typeEvaluation = Integer.parseInt(parameters.getParameter(2));
    K = 5;

    Randomize.setSeed(seed);
  }
```

2. Next, we execute the main process of the algorithm (procedure `execute`). The initial step is to abort the program if we have found a problem during the building phase of the algorithm (constructor). If everything is alright, we perform the algorithm's operations. In the case of the SGERD method we must first build the Data Base (DB) and then generate an initial Rule Base (RB). Next, the Genetic Algorithm is executed in order to find the best rules in the system. When this process is complete, we perform the final output operations. This process is shown below in its entirety (again the new inserted code is stressed in **boldface**):

```
  public void execute() {
    if (somethingWrong) {
        System.err.println("An error was found, the data-set has MV.");
        System.err.println("Please remove the examples with missing"+
            "data or apply a MV preprocessing.");
        System.err.println("Aborting the program");
    }
    else {

      dataBase = new DataBase(K, train.getnInputs(),
            train.getRanges(), train.varNames());
      ruleBase = new RuleBase(dataBase, train, typeEvaluation);
      ruleBase.initialization();

      Population pobl = new Population(ruleBase, Q, train,
            dataBase.numLabels());
      pobl.Generation();

      dataBase.saveFile(fileDB);
      ruleBase = pobl.bestRB();
      ruleBase.saveFile(fileRB);

      doOutput(val, outputTr);
```

```
      doOutput(test, outputTst);

      System.out.println("Algorithm Finished");
    }
  }
```

3. We write in an output file the DB and the RB to save the generated fuzzy model, and then we continue with the classification step for both the validation and test files. The `doOutput` procedure simply iterates all examples and returns the predicted class as a string value (in regression problems it will return a double value). This prediction is carried out in the `classificationOutput` function, which only runs the Fuzzy Reasoning Method of the generated RB (noted in **boldface**):

```
  private void doOutput(myDataset dataset, String filename) {
    String output = new String("");
    output = dataset.copyHeader();
    for(int i = 0; i < dataset.getnData(); i++) {
      output += dataset.getOutputAsString(i) + " " +
        classificationOutput(dataset.getExample(i)) + "\n";
    }
    Files.writeFile(filename, output);
  }

  private String classificationOutput(double[] example) {
    String output = new String("?");

    int clas = ruleBase.FRM(example);

    if (clas >= 0) {
      output = train.getOutputValue(clas);
    }
    return output;
  }
```

4. Finally, we show the new functions that are implemented in the myDataset class in order to obtain some necessary information from the training data during the rule learning stage. We must point out that the remaining functions of this class remain unaltered.

```
  public void computeOverlapping() {
    int i;

    classOverlapping = new double[nClasses];

    outliers = new int[nClasses];
    nExamplesClass = new int[nClasses];

    for (i = 0; i < nClasses; i++) {
      outliers[i] = nExamplesClass[i] = 0;
```

```
    }

    KNN knn = new KNN(IS, 5);
    knn.ejecutar(outliers, nExamplesClass);

    for (i = 0; i < nClasses; i++) {
      if (nExamplesClass[i] > 0) {
        classOverlapping[i] = (1.0 - (outliers[i] / nExamplesClass[i]));
      }
      else {
        classOverlapping[i] = 1.0;
      }
    }
  }

  public double getOverlapping(int nClass) {
    return (classOverlapping[nClass]);
  }
```

3.2. Integration of SGERD into KEEL Software Suite

Once the algorithm has been implemented, it can be executed directly on a terminal with the parameters file as argument. Nevertheless, when included within the KEEL software, the user can create a complete experiment with automatically generated scripts for a batch-mode execution.

In order to do so, we must recall the steps described in Section 1. Specifically, this process implies the following steps:

1. Storing the source files in the corresponding folder of the KEEL directory. We must decide the appropriate place, and since this is a Genetic Fuzzy System, the better suited will be "/src/keel/Algorithms/Fuzzy_Rule_Learni

2. Updating the build file to generate the JAR executable. Once we have decided the folder, a Manifest file must be created in order to point out the main class of the program, which should be just "Main" if we used the template. Then, we must update the "Build.xml" file in the root folder of KEEL by adding the following information:

```
<jar
    jarfile="$distMet/Clas-Fuzzy-SGERD.jar"
    manifest="$src/keel/Algorithms/Fuzzy_Rule_Learning/Genetic/ClassifierFuzzySGERD/Manifest">
    <fileset dir="$bin"
       includes="org/core/**/*.class
        keel/Algorithms/Fuzzy_Rule_Learning/Genetic/ClassifierFuzzySGERD/**/*.class
        keel/Algorithms/Preprocess/Basic/**/*.class
        keel/Dataset/**/*.class"/>
</jar>
```

3. Registering the method in KEEL. Although the JAR file can be generated and placed into the "dist" folder, the method itself will not be shown within the experimental frame if we skip this step. Basically,

we must update the proper **Master Description File** depending on the type of algorithm we are including. In the case of the SGERD method, we are working with a classification algorithm, and the XML file to be modified is the "Methods.xml" one. The information to be added is shown below:

```
<method>
    <name>SGERD-C</name>
    <family>Evolutionary Fuzzy Rule Learning</family>
    <jar_file>Clas-Fuzzy-SGERD.jar</jar_file>
    <problem_type>classification</problem_type>
    <input>
      <continuous>Yes</continuous>
      <integer>Yes</integer>
      <nominal>Yes</nominal>
      <missing>Yes</missing>
      <imprecise>No</imprecise>
      <multiclass>Yes</multiclass>
      <multioutput>No</multioutput>
    <multiinstance>No</multiinstance>
    </input>
    <output>
      <continuous>Yes</continuous>
      <integer>Yes</integer>
      <nominal>Yes</nominal>
      <missing>Yes</missing>
      <imprecise>No</imprecise>
      <multiclass>Yes</multiclass>
      <multioutput>No</multioutput>
    <multiinstance>No</multiinstance>
    </output>
  </method>
```

4. Including the configuration of the algorithm. We must point out the number of input, output, and parameters of the algorithm, as well as whether it is non-determinist algorithm, i.e. a seed is needing for the execution. This information is stored within the **Method description file** that must have exactly the same name given in the previous **Master description file**, in order to link both files. The current content for the SGERD algorithm is shown below.

```
<algorithm_specification>
  <name>Steady-state Genetic Algorithm for Extracting fuzzy
  classification Rules from Data</name>
  <nParameters>2</nParameters>
  <seed>1</seed>
  <nOutput>2</nOutput>
  <parameter>
    <name>Number of Q rules per class (0 to calculate heuristically)</name>
    <type>integer</type>
    <domain>
```

```
      <lowerB>0</lowerB>
      <upperB>40</upperB>
    </domain>
    <default>0</default>
  </parameter>
  <parameter>
    <name>Rule evaluation criteria</name>
    <type>integer</type>
    <domain>
      <lowerB>0</lowerB>
      <upperB>2</upperB>
    </domain>
    <default>2</default>
  </parameter>
</algorithm_specification>
```

5. Finally, adding the use case file. This file must also have the same name as the one given in the **Master description file**. Remember that this new file must be stored within the "dist/help" folder. Do not forget to include the most relevant information for future users.

```
<method>

    <name>Steady-State Genetic Algorithm for Extracting Fuzzy Classification Rules From Data</nam

    <reference>
<ref>E.G. Mansoori, M.J. Zolghadri, S.D. Katebi. SGERD: A Steady-State Genetic Algorithm for
Extracting Fuzzy Classification Rules From Data, IEEE Transactions on Fuzzy Systems 16:4 (2008)
1061-1071</ref>
    </reference>

    <generalDescription>

        <type>Evolutionary Fuzzy Rule Based Classification System</type>

        <objective>To extract a compact set of good fuzzy rules from numerical data</objective>

        <howWork>SGERD is a steady-state genetic algorithm, where its generations are finite
and bounded to the problem dimension. Individual selection in this algorithm is nonrandom, ...
</howWork>

        <parameterSpec>
            <param>Number of Q rules per class (0 to calculate heuristically): Max number of rule
per class. </param>
            <param>Rule evaluation criteria: User can select
              0) fCS (Aj -> Class Cj)
              1) fF (Aj -> Class Cj)
              2) f'F (Aj -> Class Cj)
</param>
        </parameterSpec>
```

```
        <properties>
            <continuous>Yes</continuous>
            <discretized>Yes</discretized>
            <integer>Yes</integer>
            <nominal>Yes</nominal>
            <valueLess>No</valueLess>
            <impreciseValue>No</impreciseValue>
        </properties>

    </generalDescription>

    <example>Problem type: Classification
Method: Clas-Fuzzy-SGERD
Dataset: iris
Training set: iris-10-1tra.dat
Test set: iris-10-1tst.dat
Test Show results: Vis-Clas-Check
Parameters: default values

After the execution of RunKeel.jar we can see into the
experiment/results/Vis-Clas-Check/TSTClas-Fuzzy-Ish-Selec folder
the classification results for the training and test sets:

TEST RESULTS
============
Classifier=
Fold 0 : CORRECT=1.0 N/C=0.0
...
```

4. Frequently Asked Questions

To conclude this document, we list several questions often raised by those developers that aimed to integrate their own methods within KEEL. This is not aimed to be an exhaustive list, but rather some additional guidelines to help researchers for the task of including new algorithms in the current version of KEEL Software Suite.

- **Q**: Should I use the API Dataset?

  A: The "myDataset" class of the template already includes the main functions for accessing the API dataset in a transparent way. Therefore, if your algorithm does not need any very specific procedure, you do not need to concern about the API Dataset.

- **Q**: How can I integrate the API Dataset into the template?

  A: In the particular case of adding new procedures within "myDataset" class that must access the API Dataset, we recommend reading the documentation of the "Instance Set" and "Attribute" classes, which are the ones that store the main information of the input data.

- **Q**: Which KEEL files may I generate (configuration, description), in order to integrate it within KEEL.
  A: The main files to be modified are the following ones: (1) "build.xml" within the root folder; (2) "Methods.xml" (or the one associated to the algorithm's family within "dist/algorithm" folder; (3) an XML file with the configuration of the algorithm itself within "dist/algorithm/methods" folder (for example); and (4) the XML use case file within "dist/help" folder. For additional information regarding this, please refer to Sections 1 and 3.2 of the current document.
- **Q**: Which files of the prototype must be updated or added in order to have the "box" of the method available within the experimental frame?
  A: Updating the **Master description file** (refer to Sections 1.5 and 3.2). Additionally, the **Method description file** must be added to indicate the method's parameters (Sections 1.1 and 3.2).
- **Q**: Should I use the "Randomize" class from KEEL as random number generator?
  A: It is not mandatory but it is recommendable as it includes the a wide amount of functions to compute random numbers with a very simple interface. Its source code is located under the "src/org/core" folder.
- Should I use the "File" class in order to implement the input/output interface with KEEL files?
  A: As in the case of the random number generator, any user can use its own class for file management. However, the "File" class has been provided in order to ease this task to the developers. Its source code is located under the "src/org/core" folder.
- **Q**: My method needs an additional library, where should I place it?
  A: The "lib" directory, which is located at the root folder, includes every single library used by KEEL. Please create a new folder to store all the "jar files" associated with the new library. Also, do not forget to add this library into the *classpath* of your algorithm, as stated in Section 1.4.
- **Q**: I have an external library to fulfill several tasks for my current algorithm. How can I access these internal procedures from the classes stored within the ".jar" file
  A: You must **import** the required classes into your source code following the class directory of the original code within the library. You must use the original path given within the jar. To do so, you can explore the content of the "jar" file. It is not necessary to include the absolute path from the KEEL directory. Finally, do not forget to add this library into the classpath, as stated previously.
- **Q**: How many inputs could have any algorithm? **A**: Any method can have as many inputs as wanted. However, user must be aware of two issues: (1) the input "box" which is connected to it must provide the correct number of outputs; and (2) the method must be prepared to read the whole number of inputs.
- **Q**: Which files are given from the KEEL interface?
  A: As stated by KEEL's constraints, at least two files will be provided, i.e. the input training and test files. However, if we are using an algorithm for a knowledge discovery task, i.e. classification,

regression, and so on, three files will be given, training, validation and test. Additionally, two output files are mandatory, i.e. training and test files. Therefore, this will be used as input of any method connected with the former. If more output files are supplied, then the "next" method in the "execution flow" must be prepared to use these files (if needed).

- **Q**: How many times does a method execute when creating the experiment?

  A: Each method is executed one time per dataset and input partition. Additionally, if the method is non-deterministic, we can select the number of executed times with different seeds within the parameters' window.