RKEEL: Using KEEL in R Code

Jose Moyano
University of Cordoba
Cordoba, Spain
Email: i02momuj@uco.es

Luciano Sanchez
University of Oviedo
Oviedo, Spain
Email: luciano@uniovi.es

***Abstract*—KEEL is a popular Java suite for a large number of different knowledge data discovery tasks. R is a emerging and quite used programming language and environment for statistical computing. In order to make a system where the advantages of KEEL and R language could be taken, it is created an R package called RKEEL, allowing to use KEEL algorithms in simple R code. The implemented R code layer between R and KEEL makes easy both using KEEL algorithms in R as implementing new algorithms for RKEEL in a very simple way. Thus allows making a more complete experimentation process.**

I. Introduction

KEEL (Knowledge Extraction based on Evolutionary Learning) [1] is an open source suite [2] implemented in Java, designed to use a large number of different knowledge data discovery tasks. One of the strengths of KEEL is that it allows to design experiments based on data flow, with different datasets and computational intelligence algorithms, in order to assess the performance of the algorithms. It contains a wide variety of classical knowledge algorithms, preprocessing techniques, computational intelligence based learning algorithms, hybrid models, statistical methodologies for contrasting experiments and so forth. So, with all these features, the user can perform a complete analysis of new computational intelligence proposals to compare to existing ones.

R [3] is a programming language and environment for statistical computing and graphics. It provides a wide variety of statistical (as linear and non-linear modelling, classical statistical tests, classification, clustering, ...) and graphical techniques. One of the principal features of R is that it can be easily extended via packages. Any user can publish a package that extends the R basic configuration. They are available principally through the CRAN (Comprehensive R Archive Network).

KEEL provides a simple GUI, useful for the design of experiments in a easy and intuitive way, but it is an inconvenient if we want to use the KEEL algorithms from source code, because KEEL does not have an API to use its functionalities. In this way, one of our objectives is to link KEEL and R, providing an interface to use KEEL features in R source code, creating the RKEEL package and making it available via CRAN repository. Thereby, with RKEEL we could preprocess the data, use algorithms from KEEL, get the results and draw a plot with them, for example, all from our R code. Also, handling with KEEL algorithms in R makes the experiment process much more complete, allowing to concatenate experiments of other R packages or other R utilities, which is a strong advantage.

The rest of the article is organized as follows: Section II presents the preliminaries, Section III shows the general structure of RKEEL, Section IV explains how to use the RKEEL package and its advantages, and Section V gives the guidelines to implement new interfaces for KEEL algorithms in RKEEL. Finally, Section VI presents the main conclusions and future work.

II. Preliminaries

In this section, first it is analysed in detail the KEEL tool, focusing on its structure and how it works. Then, it is presented RWeka, an R interface for using Weka.

A. KEEL

As mentioned before, KEEL is a suite to assess algorithms for data mining problems including regression, classification, clustering, pattern mining and so on, paying special attention to evolutionary algorithms.

The current version of KEEL (v3.0) has the following blocks:

- Data management: this part is composed by a set of tools that can be used to build new data, export and import data in other formats to KEEL format, data edition and visualization, apply transformations to data, and so on.
- Experiments: the aim of this part is the design of experiments over some selected data sets. It provides some types of experiments, as classification, regression, unsupervised learning and subgroup discovery, and a wide number of algorithms for the distinct types of experiments. Figure 1 shows an example of an experiment designed in KEEL.
- Educational experiments: this part is similar to the *Experiments* one, but it allows to debug step-by-step the experiments in order to use it as a guideline of the learning process, to use with educational objectives.
- Modules: this block implements other functionalities, as design of experiments of imbalanced learning, semi-supervised learning or multiple instance learning. Also it offers the possibility of doing non-parametric statistical analysis.

Altogether, KEEL has implemented more than 500 algorithms of multiple types, as data preprocessing, classification or regression among others.

978-1-5090-0626-7/16/$31.00 ©2016 IEEE

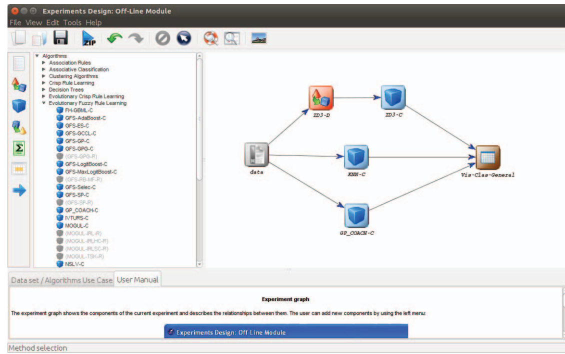

Fig. 1. Example of KEEL experiment

Once an experiment is generated and saved, KEEL creates a folder with the following structure:

- *exe*: it contains the *.jar* files of the algorithms included in the experiment. They are executed from the correspondent configuration files of each algorithm in the experiment.
- *scripts*: it contains the configuration files for each algorithm, the XML configuration file for the whole experiment and the *RunKeel.jar* file, which is used to run the experiment.
- *datasets*: it contains the dataset files required for the experiment. Each dataset is in a separately sub-folder. Also, it will store the results of the preprocessing algorithms.
- *results*: it will contain the output files of each algorithm, storing the results.

So, when the user wants to execute the experiment, he has to run the *RunKeel.jar*, which will read the configuration files and execute the corresponding experiments (by its *.jar* files) with the corresponding datasets. When each algorithm finishes, the results are stored in the *results* folder as text, and the user can read them.

Also, KEEL has a data repository on its web site where more than 900 datasets could be downloaded [4], including datasets for supervised classification, regression datasets, unsupervised learning datasets, low quality datasets and more. Most datasets can be downloaded full or in various types of partitions as 5 or 10 folds cross-validation.

B. RWeka

As similar to our proposal, it is noteworthy to mention RWeka [5]. RWeka is an R interface to Weka [6], which code is (mostly) implemented in R. Weka is a collection of machine learning algorithms for data mining tasks written in Java, containing tools for data pre-processing, classification, regression, clustering, association rules, and visualization. So, this package contains the interface code to use Weka functionalities from R code.

It uses the rJava package [7] for low-level R/Java interfacing to provide access to Weka's functionality. Thereby, the user can create R versions of Weka's classes with the usual "R look and feel", because Weka provides abstract core classes for its learners and a consistent functional methods interface for these learners classes. In conclusion, this package is a bridge between R and Weka classes to create Weka's object in R.

III. RKEEL Structure

In this section, the main structure of RKEEL is explained, focusing in its main classes and its functionality.

A. General structure of RKEEL

RKEEL is an R package, which code is completely written in R and only uses the java *.jar* executable files of KEEL. The main objective of the package is to create a layer between R and the KEEL tool, to use its functionalities in an experiment designed in R.

As mentioned before, for a designed experiment, KEEL creates a folder with all the *.jar*s and configuration files that are necessary, and then the user have to execute the *RunKeel.jar*. Thus, the main objective of our proposal is to create the full experiment folder with all the necessary *.jar* and configuration files, execute the experiment, get the results and remove the experiment folder. Thereby, in our R code, we create a learner corresponding to an algorithm, we execute it and we get the results, but the process of the experiment folder creation is transparent to the user.

Figure 2 shows the class diagram of RKEEL. As it can be seen, there is an abstract class called *KeelAlgorithm* which is the base class for all the implemented KEEL algorithms. From this class will inherit the classes that defines the different types of algorithms, as *ClassificationAlgorithm*, *RegressionAlgorithm* or *PreprocessAlgorithm*. Then, all the implemented interfaces will inherit from its corresponding class, as *KNN* or *C45* from *ClassificationAlgorithm* for example, each one declaring its parameters. In addition, they are included the classes *KeelUtils* (which contains several functions to deal with different aspects of the software, as parallelizing the experiment, dataset load, data handling and more), *ClassificationResults* (which helps to obtain some classification results from the actual and predicted classes, which are the outputs of the classification algorithms) and *RegressionResults* (to obtain and store results from a regression algorithm).

B. Dependencies

The RKEEL package has some dependences to ensure its performance. First of all, and given by KEEL, this package needs at least, Java version 7 installed on the computer. Secondly, the RKEEL package uses some R packages from the CRAN repository:

- XML [8]: it includes many approaches for both reading and creating XML documents. Version 3.98-1.3 of XML package also needs at least version 2.1 of R.
- R6 [9]: the R6 package allows the creation of classes with reference semantics, similar to R's built-in reference classes, but in a simpler and lighter-weight way, and they are not built on S4 classes so they do not require the methods package. These classes allow public and private members, and they support inheritance, even when the classes are defined in different packages. In view of these

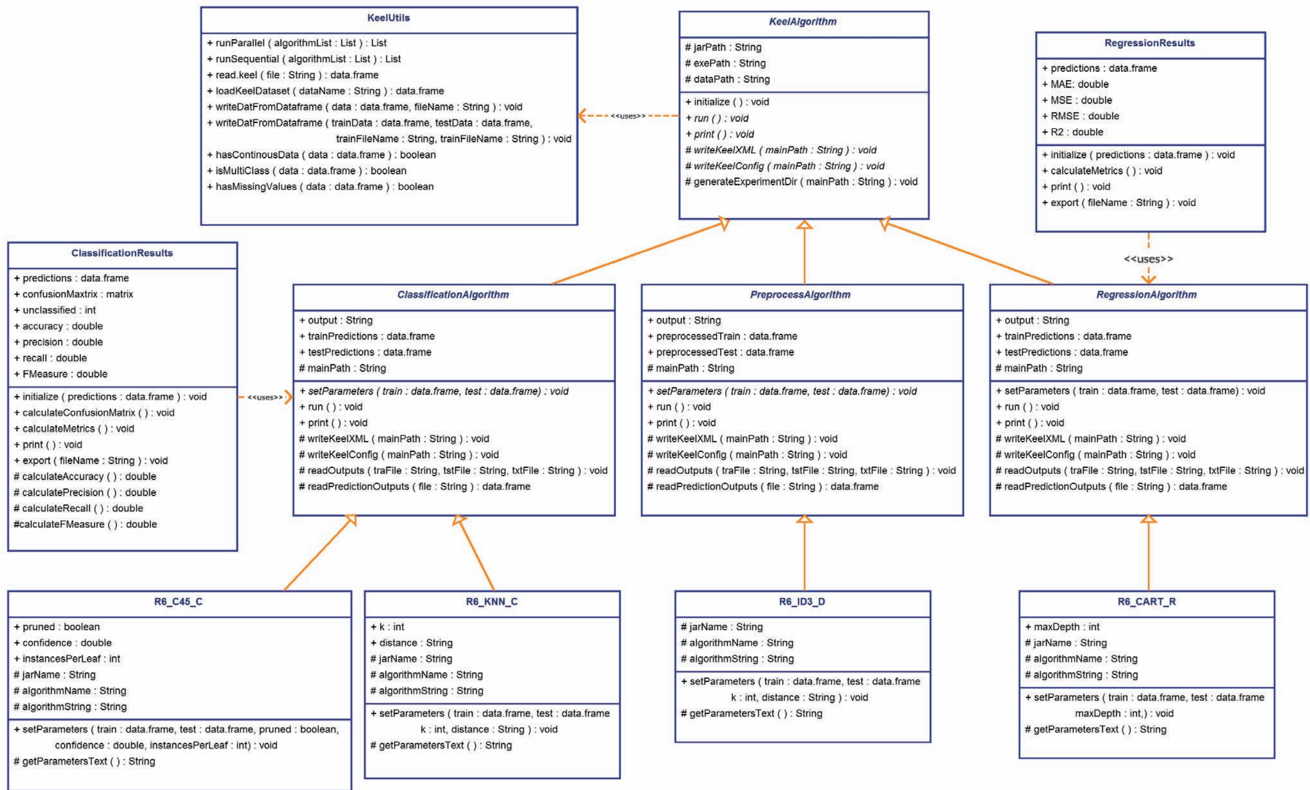

Fig. 2. RKEEL class diagram

advantages, RKEEL uses R6 classes. Version 2.1.1 of R6 package needs at least version 3.0 of R.

- doParallel [10]: this package provides a mechanism to execute *foreach* loops in parallel. Version 1.0.10 of doParallel package also needs at least 2.14 version of R.
- foreach [11]: it provides support for the foreach looping construct. Foreach is an idiom that allows for iterating over elements in a collection, without the use of an explicit loop counter. Using it also facilitates executing the loop in parallel. Version 1.4.3 of foreach package also needs at least version 2.5 of R.

Given the R version dependences of these packages, ours also needs R version 3.0 or higher.

C. Keel Algorithm

The *KeelAlgorithm* abstract class defines the main variables and functions that are necessary for any type of KEEL algorithm. Some of the functions of this class are only declared, but they must be implemented in the child ones. The main objective of this class is to have a skeleton for the child classes as well as provide some of the main utilities, as the paths of the *RunKeel.jar* file, the folder with the algorithm executables and the folder with the example datasets. Also, it is worth to mention the *generateExperimentDir* function which is responsible of the creation of the main structure of the experiment folder.

D. Classification Algorithm

Inheriting from the *KeelAlgorithm* class, *ClassificationAlgorithm* class has the common fields and functions for any classification algorithm in RKEEL. It includes variables that stores the actual and predicted classes as well as the outputs of the classifier. Regarding the functions, it includes functions to generate the XML file that describes the full experiment (*writeKeelXML*), to generate the configuration files for each algorithm (*writeKeelConfig*) and to read the classifier outputs (*readOutpts* and *readPredictionOutputs*). The *run* function is the responsible of executing the algorithm, including model training and test. The pseudo-code in Figure 3 presents the operation of this function.

Summarizing, the *run* function is the principal function of the execution, which is called then of creating the object, when we want to run the algorithm, and it is responsible of the experiment directory creation, algorithm execution and deletion of the experiment folder.

E. Classification Results

The *ClassificationResults* class is introduced in RKEEL as an extra utility, which allows to calculate some performance metrics for classification or manage the results of a classification algorithm, enriching our experiment and knowledge of the solution. When the classifier is executed, the train and test predictions are obtained from the outputs of the KEEL algorithm as an R *data.frame*. Then, we can use those

1: **procedure** RUN
2: Generate an unique folder name
3: Generate the experiment folder
4: Copy datasets to the experiment folder
5: Copy algorithm *.jar*s to the experiment folder
6: Create the results and configuration directories
7: Create XML and configuration files
8: Execute the experiment through *RunKeel.jar*
9: Read the outputs
10: Delete the experiment folder
11: **end procedure**

Fig. 3. Function *run* of *ClassificationAlgorithm* class

data.frames to create a *ClassificationResults* object, which will obtain the confusion matrix for train and test, as well as some metrics for classification performance, as accuracy, precision, recall, FMeasure or the number of unclassified instances. Also, it allows to export the results to a text file.

F. Regression Algorithm

The *RegressionAlgorithm* class inherits from *KeelAlgorithm* and, similar to *ClassificationAlgorithm*, it has the main methods and fields for any regression algorithm interface in RKEEL to be implemented.

G. Regression Results

The *RegressionResults* class is similar to *ClassificationResults*, allowing to calculate and store some performance metrics for regression algorithms as MAE, MSE, RMSE or R2. It receives the predictions of the regression algorithm, and it calculates the metrics. Also, it allows to export the results to a text file.

H. Preprocess Algorithm

The *PreprocessAlgorithm* class inherits from *KeelAlgorithm* and, it has the main methods and fields for any preprocess algorithm to be interfaced in RKEEL. After running a preprocess algorithm, they are stored two data.frame objects, which are the train and test preprocessed datasets.

I. Keel Utilities

In order to use datasets declared as *data.frames* in the RKEEL algorithms, there are some utilities implemented in *KeelUtils*. First, it is implemented the *read.keel* function, which allows to read dataset files in keel format, and returns them as an R *data.frame*. This allows RKEEL to use any dataset in *data.frame* format, as well as use the keel datasets converting them through this function. Also, as KEEL needs the datasets in keel format when executing the experiment, there are two implemented functions to convert datasets from *data.frame* to keel format: *writeDatFromDataframe* and *writeDatFromDataframes*, which allows to write respectively a single dataset or train plus test datasets at a time, to avoid some errors with the dataset classes (in some cases, test data may not contain any of the classes of the train one, which could lead to inconsistency errors if files are written separately). As KEEL includes a data repository, some of these datasets are included in RKEEL for immediate use. For this purpose, *KeelUtils* includes the *loadKeelDataset* method, which loads one of the example datasets included from KEEL and returns it in *data.frame* format. They are also included some methods to obtain data properties, as knowing if it has continuous data (*hasContinuousData*), if it has more than two classes (*isMultiClass*) or if it has missing values (*hasMissingValues*). These functions are used in algorithms which have some data restrictions, to check and ensure a proper execution.

By last, this package allows to execute several algorithms in parallel, taking full advantage of the power of the machine. For this, the function *runParallel* is implemented, which receives as argument a list with some RKEEL objects and it runs them in parallel, executing each algorithm in one core and minimizing the overall computing time with respect to the sequential implementation (which it is available as *runSequential* function).

J. Implemented algorithms

In this first version of RKEEL, they have been implemented a total of 110 interfaces to KEEL algorithms, between classification, regression and preprocess algorithms. Specifically, there are 67 classification algorithms, 15 regression algorithms and 28 preprocessing algorithms. All the implemented algorithm interfaces could be seen in the package manual. RKEEL algorithms keep the same name than KEEL ones.

K. Organization of packages

As one of the objectives was submit the package to CRAN to make it easily available, and due to restrictions and recommendations of the CRAN policies and the CRAN team, the package has been split in three different packages: *RKEELjars* (which downloads the *.jar* executable files of the different algorithms), *RKEELdata* (which includes some datasets from KEEL to use in RKEEL) and *RKEEL* (which have all the classes and functions of the package). Thereby, the RKEEL package does not exceed the size limit imposed by CRAN policies, and also has a more modular distribution.

The set of 38 datasets included from KEEL data repository are listed in Table I, showing the number of attributes (specifying how many are real (R), integer (I) or nominal (N)), the number of examples, the number of classes, the percentage of examples with missing values or the Imbalanced Ratio (IR). It includes four types of datasets: datasets for standard classification, datasets for classification with missing values, datasets for classification with imbalanced data and regression datasets.

IV. USE OF RKEEL

In this section it is introduced how to use the RKEEL package, as well as the advantages that it offers.

TABLE I
DATASETS INCLUDED IN RKEEL

Name	Attributes (R/I/N)	Examples	Classes	Missing values
Standard Classification datasets				
abalone	8 (7/0/1)	4174	28	No
banana	2 (2/0/0)	5300	2	No
bupa	6 (1/5/0)	345	2	No
car	6 (0/0/6)	1728	4	No
coil2000	85 (0/85/0)	9822	2	No
ecoli	7 (7/0/0)	336	8	No
iris	4 (4/0/0)	150	3	No
kr-vs-k	6 (0/0/6)	28056	17	No
monk-2	6 (0/6/0)	432	2	No
pima	8 (8/0/0)	768	2	No
thyroid	21 (6/15/0)	7200	3	No
tic-tac-toe	9 (9/0/0)	958	2	No
winequality-red	11 (11/0/0)	1599	11	No
winequality-white	11 (11/0/0)	4898	12	No
yeast	8 (8/0/0)	1484	10	No
zoo	16 (0/0/16)	101	7	No
Missing values datasets				
Name	**Attributes (R/I/N)**	**Examples**	**Classes**	**%MVs (examples)**
breast	9 (0/0/9)	286	2	3.15%
dermatology	34 (0/34/0)	366	6	2.19%
hepatitis	19 (2/17/0)	155	2	48.39%
mammographic	5 (0/5/0)	961	2	13.63%
marketing	13 (0/13/0)	8993	9	23.54%
mushroom	22 (0/0/22)	8124	2	30.53%
Imbalanced datasets				
Name	**Attributes (R/I/N)**	**Examples**	**Classes**	**IR**
abalone19	8 (7/0/1)	4174	2	129.44
ecoli1	7 (7/0/0)	336	2	3.36
ecoli4	8 (7/0/0)	336	2	15.8
glass	9 (9/0/0)	214	7	8.44
new-thyroid-1	5 (4/1/0)	215	2	5.14
shuttle	9 (0/9/0)	2175	7	853
winequality-red-4	11 (11/0/0)	1599	2	29.17
yeast1	8 (8/0/0)	1484	2	2.46
yeast6	8 (8/0/0)	1484	2	41.4
Regression datasets				
Name	**Attributes (R/I/N)**	**Examples**		**Missing values**
AutoMPG6	5 (2/3/0)	392		No
delta_elv	5 (5/0/0)	9517		No
diabetes	2 (2/0/0)	43		No
forestFires	12 (7/5/0)	517		No
friedman	5 (5/0/0)	1200		No
mv	10 (7/3/0)	40768		No
plastic	2 (2/0/0)	1650		No

A. Installation

The RKEEL package is available through the CRAN repository, and it can be downloaded via its GitHub repository (https://github.com/i02momuj/RKEEL) too. To install it, the steps of Figure 4 have to be followed. Thereafter, we can start using RKEEL. It is only necessary to install RKEEL package, RKEELdata and RKEELjars packages will be installed automatically as dependencies.

```
> #Install RKEEL from CRAN Repository
> install.packages("RKEEL")
> library(RKEEL)
```

Fig. 4. Installing RKEEL package

B. Loading data

One of the strengths of the RKEEL package is that we can chain the KEEL experimentation with R statistical tests, draw plots with the results of a KEEL algorithm, use datasets from KEEL in experimentations in R, and so, i.e. join de goodness of R and KEEL. RKEEL has a simple user-friendly interface that makes creating experiments in a very easy way. First, it is remarkable that, as RKEEL datasets must be *data.frame* data, multiple types of dataset can be used, since R includes some datasets and some functions to read dataset files as *read.csv* and *read.arff* among others. In this way, RKEEL implements a *read.keel* function, to read dataset files in KEEL format and also has some datasets included. Figure 5 shows different ways to load a dataset to use it in RKEEL, definitely, any dataset in *data.frame* format.

```
> #Load R dataset
> data(iris)
>
> #Read .arff file
> iris_arff_train <- read.arff("mydata/iris-tra.arff")
> iris_arff_test <- read.arff("mydata/iris-tst.arff")
>
> #Read .dat dataset (keel format)
> iris_train <- read.keel("mydata/iris-tra.dat")
> iris_test <- read.keel("mydata/iris-tst.dat")
>
> #Load RKEEL dataset
> iris_RKEEL <- loadKeelDataset("iris")
```

Fig. 5. Loading data for RKEEL use

C. Creating and running an algorithm

To create a RKEEL algorithm object, it must be used the algorithm name, passing its parameters in parentheses. This function needs the train and test *data.frame* and the parameters of the algorithm, as shown in Figure 6. If any parameter value is not indicated, it uses its default value. Then, to run the algorithm, we have to use the *run* function of the object.

```
> #Create algorithm object with default values
> learner_C45 <- C45_C(iris_train, iris_test)
>
> #Show object
> learner_C45
----------------------------------
C4.5 Decision Tree
----------------------------------
pruned = TRUE
confidence = 0.25
isntancesPerLeaf = 2
----------------------------------
>
> #Run a single algorihtm
> learner_C45$run()
Algorithm executed successfully
```

Fig. 6. Creating and running a C45 classifier with RKEEL

Alike, to run a preprocessing algorithm from RKEEL, the steps in Figure 7 have to be followed.

D. Running algorithms in parallel

Whenever instead of running a single algorithm we want to run several into an experiment, RKEEL implements the *runParallel* function, which allows to run a group of algorithms

```
> #Create a preprocess algorithm
> pre_ID3 <- ID3_D(iris_train, iris_test)
> pre_ID3$run()
>
> #Execute a classifier with the preprocessed datasets
> learner_ID3 <- ID3_C(pre_ID3$preprocessed_train,
    pre_ID3$preprocessed_test)
> learner_ID3$run()
Algorithm executed successfully
```

Fig. 7. Creating and running a preprocessing algorithm with RKEEL

in parallel in a machine. This function receives as arguments a list with RKEEL algorithm objects, and the number of cores to parallelize (if the number of cores is not specified, it detects them automatically), executes the algorithms in parallel in the specified cores and returns a list with the executed algorithms. Figure 8 shows an example of how to run the algorithms in parallel. This is very helpful when it is needed to know the performance of a lot of algorithms or an algorithm with multiple configurations.

```
> #Run a group of algorithms in parallel
> #Create some classifiers learner1, ... learnerN
> executedAlgs <- runParallel(list(learner1, ...,
    learnerN))
Executing experiment in 4 cores
Experiment executed successfully
```

Fig. 8. Running algorithms in parallel in RKEEL

In order to show the improvement in runtime when using the *runParallel* function, we have done an experiment where an R script with 10 KNN algorithms with different parameters are executed (with $K = 1, 3, 5, 7$ *and* 9; and $distance =$ "$Euclidean$" *and* "$Manhattan$") in 4 datasets, as example of tuning parameters experimentation. They have been executed sequentially and in parallel, measuring the execution time with *system.time()* function. As shown in Table II, the parallel execution is more than 3x faster on average than sequential, using 4 cores in parallel instead of 1 in sequential manner. It is noteworthy that the more complex are the experiments, greater is the improvement of the parallelization (the datasets are ordered by complexity in the table, taking as complexity the product of number of attributes and number of instances). This speed-up is a clear advantage of the package, because offers the possibility of reducing the execution time of an experiment significantly.

TABLE II
PARALLEL EXECUTION IMPROVEMENT

Dataset	Sequential time (s)	Parallel time [4 cores] (s)	Speed-up
marketing	206.46	67.25	3.070
thyroid	322.27	98.72	3.264
mushroom	182.80	56.23	3.251
coil2000	1786.593	476.786	3.747
		Mean	**3.329**

E. Obtaining results

When the algorithm finishes running, in a *Classification-Algorithm* for example, the fields *trainPredictions* and *test-Predictions* stores each one the actual and predicted class for each dataset. In order to obtain more classification results, we can create a *ClassificationResults* object with the *new* function, passing the prediction as parameter (Figure 9). Thus, we can extend our results from the classifier prediction to a set of metrics for measuring the classifier performance. Besides, it is possible to export the results to a text file with the *export* function. The same could be done with the results of a *RegressionAlgorithm*, creating a *RegressionResults* object.

```
> #Print classifier predictions
> learner_C45$testPredictions
        Real    Predicted
1      setosa       setosa
2      setosa       setosa
  ...
43  virginica   versicolor
44  virginica    virginica
45  virginica    virginica
>
> #Calculate some classification results
> C45_trainResults <- ClassificationResults$new(
    learner_C45$trainPredictions)
> C45_testResults <- ClassificationResults$new(
    learner_C45$testPredictions)
>
> #Print test results
> C45_testResults
Confusion matrix:
           setosa versicolor virginica
setosa         14          1         0
versicolor      0         15         0
virginica       0          1        14

 Unclassified istances: 0

Metrics:
   Accuracy: 0.9555556
   Precision: 0.9607843
   Recall: 0.9555556
   FMeasure: 0.9581628
>
> #Export the results
> C45_testResults$export("C45_results.txt")
```

Fig. 9. Obtaining classification results from C45 classifier

F. Joint experimentation

As it has been detailed, it is too easy to create an experimentation with the RKEEL environment, in only four steps: load the dataset, create the algorithm object, run the algorithm and get the results. But, further than just use KEEL functions in R, the strength of this package is that we can join these algorithms with other functionalities of R such data preprocessing or drawing plots, or use other algorithms as those included in the RWeka package.

The example of Figure 10 shows an experimentation where it looks over some decision trees (with default parameter values) to select that which performs better for a given dataset. The dataset is loaded from RKEEL and then it is preprocessed with R. It runs and compares algorithms from both RKEEL and RWeka. When the execution finishes, it plots a bar chart

with the percentage of correct classified instances for each classifier through R (Figure 11) in order to see in a graphic way which is the best. Also, for the best classifier, as shown in Figure 12, it is plotted the actual and predicted classes for each instance and two of the data features. That example emphasizes that RKEEL gives the possibility of making a much more complete experimentation process.

```
> #Load KEEL dataset
> iris <- loadKeelDataset("iris")
>
> #Stratify dataset with R
> library(caTools)
> train_rows <- sample.split(iris$Class, SplitRatio=0.7)
> iris_train <- iris[which(train_rows),]
> iris_test <- iris[-which(train_rows),]
>
> #Create RWeka classifiers
> learner_J48 <- RWeka::J48(Class~., data = iris_train)
> results_J48 <- RWeka::evaluate_Weka_classifier(
    learner_J48, newdata=iris_test)
> #Create LMT and DecistionStump from RWeka too
>
> #Create RKEEL classifiers
> learner_C45 <- RKEEL::C45$new(iris_train, iris_test,
    TRUE, 0.25, 2)
> #Create AdaBoosNC, C45Binarization, CART, DT_GA and
    PUBLIC from RKEEL too
>
> #Run RKEEL classifiers in parallel
> exec_algs <- RKEEL::runParallel(algorithmList)
> learner_C45 <- exec_algs[[1]]
> results_C45 <- RKEEL::ClassificationResults$new(
    learner_C45$testPredictions)
> #Get the other results too
>
> #Create a vector with the accuracy for each algorithm
>
> #Plot the accuracy for each classifier
> barplot <- barplot(accuracyVector, space = 0.8, xlab =
     "CLASSIFIER", ylab = "CCR", axes = TRUE, ylim = c
    (0, 100))
```

Fig. 10. Example of joint experimentation with RKEEL and RWeka

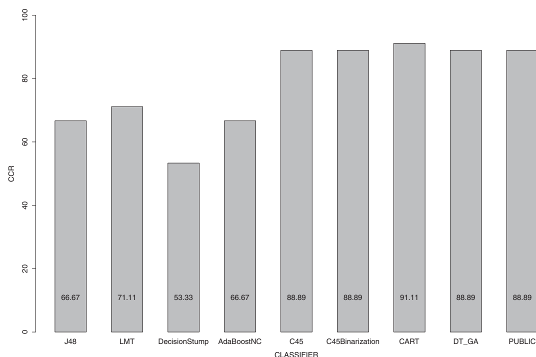

Fig. 11. Bar Chart with CCR for each classifier

When comparing RKEEL and RWeka, it can be observed that RKEEL is more complete than RWeka in terms of implemented algorithms, while RKEEL has 110 algorithms altogether, RWeka has only 35 algorithms implemented. This indicates that RKEEL offers a new perspective and strengths

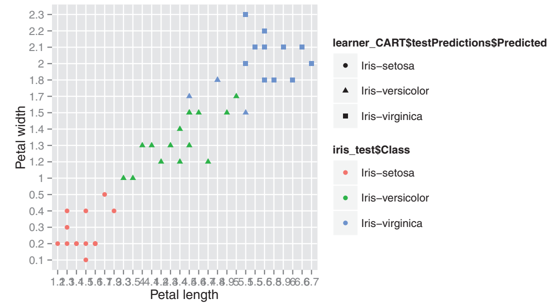

Fig. 12. Plot with Iris classification results

when creating experiments in R, and when doing a joint experimentation RKEEL allows to extend the classifier algorithms available if only using RWeka. Moreover, RKEEL provides the ability of running algorithms in parallel, which is not available for RWeka algorithms. Also, the RKEEL interface is simpler, where to create a classifier, only the datasets and the parameters have to be introduced.

V. Implementing new algorithms

The RKEEL package has been implemented so that including a new algorithm is very easy. It allows the user to expand the package with new algorithm interafces in only few steps. Figure 13 shows an example of how to create one new algorithm in RKEEL. First, we have to create a R6 class that inherits from the *ClassificationAlgorithm* class (lines 3-8). The class name should be the algorithm name in KEEL, preceded by "R6", to ensure consistency between RKEEL algorithms and KEEL ones. Then, in the public side, we declare the different variables corresponding to the parameters of the algorithm (lines 10-14) and a *setParameters* function which is the responsible of providing values to the parameters (lines 17-33). The *setParameters* function first has to call to the same function of the super class, then if necessary checks some data restrictions, and finally gets the parameter values. In the private side of the class, it must be declared 3 variables: the name of the executable *.jar* of the algorithm (this *.jar* must be placed in the folder specified as *jarPath*), the algorithm name in KEEL and the algorithm string found in the configuration file of the KEEL algorithm (lines 34-44). Then, we have to declare the *getParametersText* function (lines 46-55), which will return a string with the parameters as seen in the KEEL configuration file for that algorithm. This information is also included when printing the object. Finally, and out of the R6 Class, it is declared an alias for an easier use and allowing a better documentation of the package (lines 59-64). The name have to be the same of the class but without the "R6". Inside the function, the algorithm is created with the *new* function, the parameters are setted, and the algorithm is returned.

```r
#Guidelines to create a new classificationAlgorithm

#R6 Class implementing the algorithm
R6_Dummy_C <- R6::R6Class("R6_Dummy_C",
#Replace Dummy_C with the class name

  #Inherit from classification algorithm
  inherit = ClassificationAlgorithm,

  public = list(

    #Algorithm parameters separated with commas
    param1 = 3,
    param2 = TRUE,

    #setParameters function
    setParameters = function(train, test, param1,
        param2){

      #Call the super$setParameters function
      super$setParameters(train, test)

      #Check for any data constraints
      if((isMultiClass(train))||(isMultiClass(test))){
        stop("Does not accept multi-class dataset.")
      }

      #Assign parameters value
      self$param1 <- param1
      self$param2 <- param2
    }

  ),

  private = list(

    #jar Filename
    jarName = "Dummy.jar",

    #algorithm name in KEEL
    algorithmName = "Dummy-C",

    #String with algorithm name
    #(in first line of configuration file)
    algorithmString = "Dummy Classifier",

    #Get the text with the parameters for config file
    getParametersText = function(){

      #Parameters list as in KEEL configuration file
      text <- ""
      text <- paste0(text, "P1 = ", self$param1, "\n")
      text <- paste0(text, "P2 = ", self$param2, "\n")

      return(text)
    }
  )
)

#Create alias for easier use and better documentation
Dummy_C <- function(train, test){
  alg <- RKEEL::R6_Dummy_C$new()
  alg$setParameters(train, test)
  return (alg)
}
```

Fig. 13. Example of RKEEL algorithm implementation

VI. Conclusion

We have implemented an R package called RKEEL, which is available at CRAN repository. With this package, it is possible to run the KEEL algorithms from R code, making the experimentation process much more complete, since RKEEL includes 110 algorithms from KEEL. This allows to join the KEEL experiments with other algorithms available in R, as RWeka. Also, it is possible to run the RKEEL algorithms in parallel, optimizing the execution time of the experiment. Besides, adding a new algorithm to RKEEL is too easy, only following a few steps. We have shown the package structure, how to implement new algorithm interfaces and the main advantages, focusing in enriching both R and KEEL by joining their advantages in a single experimentation.

As future work, we plan to implement more KEEL algorithms, including more classification, regression and preprocess algorithms as well as other types as unsupervised learning, and implementing other functionalities such as running a k-fold cross-validation. In addition, as RKEEL offers the possibility of running algorithms in parallel in a single machine we intend to make a distributed version of the parallel function, to let parallelize the algorithms in several machines.

Acknowledgment

This work was supported by the Spanish Ministry of Science and Innovation (MICINN) and the Regional Ministry of the Principality of Asturias under Grants TIN2014-56967-R and FC-15-GRUPIN14-073.

References

[1] J. Alcala-Fdez, L. Sanchez, S. Garcia, M. del Jesus, S. Ventura, J. Garrell, J. Otero, C. Romero, J. Bacardit, V. Rivas, J. Fernandez, and F. Herrera, "Keel: A software tool to assess evolutionary algorithms for data mining problems," *Soft Computing*, vol. 13, no. 3, pp. 307–318, 2009.

[2] J. Alcala-Fdez and J. M. Alonso, "A Survey of Fuzzy Systems Software: Taxonomy, Current Research Trends, and Prospects," *IEEE TRANSACTIONS ON FUZZY SYSTEMS*, vol. 24, no. 1, pp. 40–56, 2016.

[3] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2015. [Online]. Available: https://www.R-project.org/

[4] J. Alcala-Fdez, A. Fernandez, J. Luengo, J. Derrac, S. Garca, L. Snchez, and F. Herrera, "Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework," *Journal of Multiple-Valued Logic and Soft Computing*, vol. 17, no. 2-3, pp. 255–287, 2011.

[5] K. Hornik, C. Buchta, and A. Zeileis, "Open-source machine learning: R meets Weka," *Computational Statistics*, vol. 24, no. 2, pp. 225–232, 2009.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations*, vol. 11, no. 1, 2009.

[7] S. Urbanek, *rJava: Low-Level R to Java Interface*, 2015, r package version 0.9-7. [Online]. Available: http://CRAN.R-project.org/package=rJava

[8] D. T. Lang and the CRAN Team, *XML: Tools for Parsing and Generating XML Within R and S-Plus*, 2015, r package version 3.98-1.3. [Online]. Available: http://CRAN.R-project.org/package=XML

[9] W. Chang, *R6: Classes with Reference Semantics*, 2015, r package version 2.1.1. [Online]. Available: http://CRAN.R-project.org/package=R6

[10] R. Analytics and S. Weston, *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*, 2015, r package version 1.0.10. [Online]. Available: http://CRAN.R-project.org/package=doParallel

[11] ——, *foreach: Provides Foreach Looping Construct for R*, 2015, r package version 1.4.3. [Online]. Available: http://CRAN.R-project.org/package=foreach