Kurt Hornik and Christian Buchta and Achim Zeileis
Open-Source Machine Learning: R Meets Weka

*Original Citation:*

Hornik, Kurt [ORCID: https://orcid.org/0000-0003-4198-9911](https://orcid.org/0000-0003-4198-9911) and Buchta, Christian and Zeileis, Achim
(2007)

This version is available at: [https://epub.wu.ac.at/1188/](https://epub.wu.ac.at/1188/)
Available in ePub\(^\text{WU}\): March 2007

ePub\(^\text{WU}\), the institutional repository of the WU Vienna University of Economics and Business, is provided by the University Library and the IT-Services. The aim is to enable open access to the scholarly output of the WU.
Open-Source Machine Learning: R Meets Weka

Kurt Hornik, Christian Buchta, and Achim Zeileis
Wirtschaftsuniversität Wien

Abstract

Two of the prime open-source environments available for machine/statistical learning in data mining and knowledge discovery are the software packages Weka and R which have emerged from the machine learning and statistics communities, respectively. To make the different sets of tools from both environments available in a single unified system, an R package RWeka is suggested which interfaces Weka’s functionality to R. With only a thin layer of (mostly R) code, a set of general interface generators is provided which can set up interface functions with the usual “R look and feel”, re-using Weka’s standardized interface of learner classes (including classifiers, clusterers, associators, filters, loaders, savers, and stemmers) with associated methods.

Keywords: machine learning, statistical learning, Weka, R, Java, interface.

1. Introduction

New Zealand has brought forth two kinds of wekas: a flightless endemic bird (Gallirallus australis) and the Waikato Environment for Knowledge Analysis (Weka, http://www.cs.waikato.ac.nz/~ml/weka/), the leading open-source project in machine learning. Weka is a comprehensive collection of machine-learning algorithms for data mining tasks written in Java and released under the GPL, containing tools for data pre-processing, classification, regression, clustering, association rules, and visualization. There are three graphical user interfaces (“Explorer”, “Experimenter” and “KnowledgeFlow”) as well as a standardized command line interface. The Weka project was started in 1992, and has been funded by the New Zealand government since 1993. It has recently joined Pentaho (http://www.pentaho.com/), a leading and award-winning open-source business intelligence project, to add “data mining capabilities to the broad range of business intelligence features” of Pentaho.

Weka complements the book “Data Mining” (Witten and Frank 2005) which is heavily used in computer science curricula. It implements a variety of methods popular in machine learning and useful for statistical learning, but typically not available in statistical software packages. This includes rule (JRip), lazy (LBR), and meta learners (MultiBoostAB), as well as cluster algorithms such as CobWeb and DBSCAN, or the association rule algorithm Tertius. For many algorithms, Weka provides de-facto reference implementations, including the key decision tree algorithms J4.8 and M5’ implementing C4.5 and M5, respectively. See Witten and Frank (2005) for more details and references. Finally, Weka also serves as the basis for a variety of additional machine learning software projects.

Obviously, it is highly desirable that statisticians have convenient and efficient access to Weka’s functionality, ideally through seamless integration into their commonly employed software environment. This paper discusses a Weka interface for R (R Development Core Team 2007), the leading open-source system for statistical computing and graphics, which is provided by the R extension package RWeka (Hornik, Zeileis, Hothorn, and Buchta 2007). Section 2 presents the interfacing methodology employed. Section 3 discusses limitations and possible extensions, which also relate to general issues arising when interfacing R with “foreign” (e.g., Java-based) systems.
2. Interfacing Weka to R

There are several design issues which relate to the choice of the interface approach taken, including generalizability (if access is desired only to a restricted subset of the available functionality, hand-crafted interface functions suffice) and maintainability (if the foreign system is modified for interfacing purposes, patches need to be maintained along with new releases). At the technology level, a system such as Weka can be interfaced “directly” via the operating system’s access to the command line interface or by building on low-level R/Java interfaces, such as rJava (Urbanek 2007), SJava (Temple Lang and Chambers 2005), or arji (Carey 2007). At the user level, one could create R versions of Weka’s classes and an object-oriented programming (OOP) style interface for Weka’s methods (typically by writing $ methods, i.e., “overloading” the $ operator in OOP jargon).

Package RWeka builds on package rJava for low-level direct R/Java interfacing to provide access to Weka’s core functionality. As Weka provides abstract “core” classes for its learners as well as a consistent “functional” methods interface for these learner classes, it is possible to provide general interface generators that re-use Weka methods. These yield R functions and methods with “the usual look and feel”, e.g., a customary formula interface for supervised learners (which are called “classifiers” in Weka’s terminology), again by re-using corresponding Weka methods. This approach allows for both generalizability (because new interfaces can be generated on the fly) as well as maintainability (because only the “exported” functionality of Weka is re-used). In the following, setting up and fitting classifiers is discussed in more detail—Table 1 gives an overview of the R/RWeka functions/methods and their Weka counterparts.

RWeka contains R classes I (interface classes) for each key “group” of functionality provided by Weka and to be interfaced (currently, classifiers, clusterers, associators, filters, loaders, savers, and stemmers), and functions mI (interface generators) which generate such interfaces by returning suitably classed functions fI,W interfacing given Weka classes W. The interface functions fI,W have formals “as usual” and are suitably classed so that standard R methods can be provided. The implementation is based on the S3 object system (Chambers and Hastie 1992). The mechanism is best illustrated by an example:

R> library("RWeka")
R> foo <- make_Weka_classifier("weka/classifiers/trees/J48", c("bar","Weka_tree"))

The interface generator make_Weka_classifier() (mI) creates an interface function foo() (fI,W) to the given Weka class ‘weka.classifiers.trees.J48’ (W) whose fully qualified class name is specified in JNI notation. The interface function foo() in fact inherits from the interface class ‘R_Weka_classifier_interface’ (I). When fitted to data sets, it returns objects inheriting from the given classes ‘bar’ and ‘Weka_tree’ as well as ‘Weka_classifier’ which objects returned by classifier interface functions always inherit from. All classifier interface functions have the usual formals formula, data, subset and na.action, as well as formal control for specifying control arguments to be passed to Weka (in this case, when building the classifier). Printing such interface functions uses Weka’s globalInfo() and technicalInformation() methods to provide a description of the functionality being interfaced.

R> print(foo)

An R interface to Weka class ‘weka.classifiers.trees.J48’, which has information

Class for generating a pruned or unpruned C4.5 decision tree. For more information, see

Argument list:
foo(formula, data, subset, na.action, control = Weka_control())

Returns objects inheriting from classes:
bar Weka_tree Weka_classifier

When the classifier interface function is called, a model frame is set up in R which is transferred to a Weka instance object. Then, the buildClassifier() method of the Weka class interfaced is called with these instances. The fitted values (model predictions for the training data) are obtained by calling the Weka classifyInstances() method for the built classifier and each training instance. As an example, a J4.8 tree for the iris data can be grown via

R> fm <- foo(Species ~ ., data = iris, control = Weka_control(S = TRUE, M = 5))

J48 pruned tree
------------------
Petal.Width <= 0.6: setosa (50.0)
Petal.Width > 0.6
  | Petal.Width <= 1.7
  |  | Petal.Length <= 4.9: versicolor (48.0/1.0)
  |  | Petal.Length > 4.9: virginica (6.0/2.0)
  | Petal.Width > 1.7: virginica (46.0/1.0)

Number of Leaves :  4
Size of the tree :  7

A suitably classed object containing both a reference to the built classifier and the predictions is returned. Such objects have at least a print() method (using Weka’s toString()), a summary() method (using Weka’s ‘Evaluation’ class), and a predict() (and fitted()) method for either “classes” (again using Weka’s terminology: numeric for regression, factor for classification) or class probabilities (using Weka’s distributionForInstance()). Therefore, a confusion matrix can easily be computed “by hand” in R or re-using Weka’s functionality via the summary() method (which also prints further summary statistics).

R> table(observed = iris$Species, predicted = fitted(fm))
observed    predicted
           setosa versicolor virginica
setosa     50        0        0
versicolor  0       47        3
virginica   0        1        49
R> summary(fm)

### Summary ###

Correctly Classified Instances 146 97.3333 %
Incorrectly Classified Instances 4 2.6667 %
Kappa statistic 0.96
Mean absolute error 0.0293
Root mean squared error 0.1209
Relative absolute error 6.5815 %
Root relative squared error 25.6545 %
Total Number of Instances 150

### Confusion Matrix ###

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>47</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>49</td>
<td>49</td>
</tr>
</tbody>
</table>

Weka provides command-line style options for controlling building the classifiers. These can be queried online using WOW(), the Weka Option Wizard (taking advantage of Weka's `listOptions()`). The desired control options can be given using the `control` argument of the interface function using `Weka_control()`. This allows the user to conveniently employ R's typical tag-value style (`(S = TRUE, M = 5)`) which is internally wrapped to Weka's command-line option style (e.g., `'-S -M 5'`). In the J4.8 example above, the control arguments were set to build a J4.8 tree without subtree raising (`S = TRUE`) and setting the minimal number of instances per leaf to 5 (`M = 5`).

R> WOW(foo)

- **U**: Use unpruned tree.
- **C**: Set confidence threshold for pruning. (default 0.25) Number of arguments: 1.
- **M**: Set minimum number of instances per leaf. (default 2) Number of arguments: 1.
- **R**: Use reduced error pruning.
- **N**: Set number of folds for reduced error pruning. One fold is used as pruning set. (default 3) Number of arguments: 1.
- **B**: Use binary splits only.
- **S**: Don't perform subtree raising.
- **L**: Do not clean up after the tree has been built.
- **A**: Laplace smoothing for predicted probabilities.
- **Q**: Seed for random data shuffling (default 1). Number of arguments: 1.

In addition to classifiers, RWeka provides interface generation facilities for clusterers, associators, filters, loaders, savers, and stemmers, with filter interface functions also exhibiting a formula-style interface. For some of the most important algorithms—but not for all—interface functions are readily provided. Users can employ the interface generator functions to create additional interface functions at their discretion, or even create interface functions different from the default ones, typically to modify the return signature to feature dispatch to different, potentially user-defined, `plot()` or `summary()` methods.
The generality of the RWeka approach is made possible by the fact that, for the kinds of functionality interfaced, Weka provides abstract “core” classes (e.g., ‘weka.classifiers.Classifier’) with basic methods (e.g., buildClassifier() or classifyInstance()) as well as standardized interfaces such as OptionHandler or TechnicalInformationHandler, such that key functionality can be accessed in uniform ways. There are situations, however, where interface computations need to be specialized. For example, Weka’s meta learners expect the base learner to be given with their fully classified Java class name, but R users would naturally like to specify the interface functions (or at least only the “base names” of the Java classes). Thus, for interfaces to meta learners, the control options should be rewritten accordingly. RWeka uses the notion of handlers for these situations, which are named lists of functions to be called for certain purposes. Currently, options handlers are used by classifier and saver interfaces, and the corresponding interface generators allow their specification.

The specification of the return signature of the interface functions allows dispatching to specialized S3 methods. As one example, package RWeka provides a customized plot() method for the models returned by the Weka tree learners (such as J48(), M5P() or LMT()) which inherit from class ‘weka.tree’. This method is based on the routines for plotting ‘BinaryTree’ objects in package party (Hothorn, Hornik, and Zeileis 2006). For Weka learners implementing the Drawable interface, i.e., providing a graph() method, it is also possible to use write_to_dot() to create DOT language representations of the built classifiers for processing via the dot program of Graphviz (Ellson, Gansner, Koutsofios, North, and Woodhull 2003). These approaches are illustrated in the code below, yielding the visualizations shown in Figure 1.

R> plot(fm)
R> write_to_dot(fm, "iris-J48.dot")
R> system("dot -Tps iris-J48.dot > iris-J48.eps")

Additionally, the DOT representation could be read back into R and visualized by means of the Rgraphviz package (Gentry and Long 2007). In fact, one could also try to interface Weka’s native plotting facilities. This is currently not done, as it cannot easily be integrated into R’s device system, and thus would not comply with the design principle of remaining within R’s “usual look and feel”.

The RWeka code base consists of two components in addition to the Weka jar file (unmodified to maximize maintainability). The major component is high-level R code for interface generators and reporters, and useful methods, based on the low level R/Java interface provided by package rJava. In addition, there is some Java-level interface code for enhancing performance. E.g., unlike R’s data frames, Weka’s instance objects are organized row-wise, and predictions (using

![Figure 1: Visualizing fitted Weka trees within R and via Graphviz.](image-url)
3. Discussion

We see three directions for possible enhancements of the current functionality of RWeka, which relate to general issues arising when interfacing R with other systems.

Too much privacy: By definition, information private to Weka’s classes is not available for interfacing. E.g., for fitted tree learners, Weka provides a description of the tree in the DOT language. However, this contains the chosen split variables and split points only as character strings which can not be re-used (e.g., for computing predictions within R) unless this string representation is reverse engineered. Similar problems occur for other models (e.g., for LinearRegression from which the terms structure of the AIC-selected model cannot be extracted). If by design a system interfaced is not modified for interfacing purposes (as in our case), then such issues need to be resolved at the upstream source level. We have begun to work with the Weka developers to add functionality typically available in state-of-the-art statistical software. E.g., Weka 3.5.4 has added a getAllTheRules() method to access the association rules found by its Tertius or Apriori implementations, allowing for efficient integration of these algorithms into the association rule mining environment provided by package arules (Hahsler, Grün, and Hornik 2005).

Too much data manipulation: E.g., when using a filter interface function such as Discretize(), data available as an R data frame are transformed to Weka instances, filtered, and transformed back to a data frame. If the next data analysis step again employs an RWeka interface function, some of these data transformations are unnecessary. A natural idea would be having common R/Java data objects encapsulating data access for both systems in a way that transformations between native representations are only performed when needed (e.g., as references with transform-on dereference semantics). A somewhat simpler and less symmetric approach would try to employ suitably classed R objects which when evaluated transfer data back into R, but can be dispatched upon without evaluation. However, this is not possible given R’s current semantics. Developing efficiently and transparently designed proxy objects for common data handling is a general issue when interfacing two systems (e.g., the various data base interfaces available for R).

One-way communication: The current R/Weka interface is entirely asymmetric in nature as there is no way to access R’s functionality from the Weka side. Such “callbacks” could be useful in a variety of circumstances, e.g., to employ R classifiers as base learners for Weka’s meta learners, or a user-defined R dissimilarity measure as the distance function used by Weka’s clusterers. One idea
would be creating Weka classes representing the corresponding R functionality (and ideally extending one of Weka’s abstract classes) and providing the basic methods (e.g., `buildClassifier()`, `classifyInstance()` for classifiers representing R regression or classification models) by calling back into R. Ideally, such Weka-to-R interface classes would be created using an interface generation approach along the lines described in Section 2. However, apart from implementation issues such as threading disparity, it is currently unclear whether such callbacks can be implemented in a satisfactorily efficient way: Clearly, efficient data sharing across systems as discussed above is a key prerequisite.

**References**


