RWeka Odds and Ends

Kurt Hornik

September 3, 2024

RWeka is an R interface to Weka (Witten and Frank, 2005), 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. Building on the low-level R/Java interface functionality of package **rJava** (Urbanek, 2016), **RWeka** provides R "interface generators" for setting 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. Hornik, Buchta, and Zeileis (2009) discuss the design philosophy of the interface, and illustrate how to use the package.

Here, we discuss several important items not covered in this reference: Weka packages, persistence issues, and possibilities of using Weka's visualization and GUI functionality.

1 Weka Packages

On 2010-07-30, Weka 3.7.2 was released, with a new package management system its key innovation. This moves a lot of algorithms and tools out of the main Weka distribution and into "packages", featuring functionality very similar to the R package management system. Packages are provided as zip files downloadable from a central repository. By default, Weka stores packages and their information in the Weka home directory, as specified by the environment variable `WEKA_HOME`; if this is not set, the 'wekafiles' subdirectory of the user's home directory is used. Inside this directory, subdirectory 'packages' holds installed packages (each contained its own subdirectory), and 'repCache' holds the cached copy of the meta data from the central package repository. Weka users can access the package management system via the command line interface of the `weka.core.WekaPackageManager` class, or a GUI. See e.g. `https://waikato.github.io/weka-wiki/packages/manager/` for more information.

For the R/Weka interface, we have thus added `WPM()` for manipulating Weka packages from within R. One starts by building (or refreshing) the package metadata cache via

```
> WPM("refresh-cache")
```

and can then list already installed packages

```
> WPM("list-packages", "installed")
```

or list all packages available for additional installation by

```
> WPM("list-packages", "available")
```

Packages can be installed by calling `WPM()` with the `"install-package"` action and the package name, and similarly be removed using the `"remove-package"` action. Finally, packages can be "loaded" (i.e., having their jars added to the class path) using the `"load-packages"` action.

Note that if the Weka home directory was not created yet, `WPM()` will instead use a temporary directory in the R session directory: to achieve persistence, users need to create the Weka home directory before using `WPM()`.

The advent of Weka's package management system adds both flexibility and complexity. Package **RWeka** not only provides the "raw" interface generation functionality, but in fact registers

interfaces to the most commonly used Weka learners, such as J4.8 and M5'. Some of these learners were moved to separate Weka packages. For example, the Weka classes providing the Lazy Bayesian Rules classifier interfaced by `LBR()` are now in Weka package **lazyBayesianRules**. Hence, when `LBR()` is used for building the classifier (or queried for available options via `WOW()`), the Weka package must be loaded (and hence have already been installed). As of **RWeka** 0.4-32, the interface registration mechanism provides a `package` argument to optionally specify an external package providing a Weka learner class being interfaced, to the effect that that this package is loaded before the class is instantiated. E.g., the **RWeka** registration code now does

```
> LBR <-
+     make_Weka_classifier("weka/classifiers/lazy/LBR",
+                          c("LBR", "Weka_lazy"),
+                          package = "lazyBayesianRules")
```

(Previously, the `init` argument would be used for explicit loading, so that for example the above registration had `init = make_Weka_package_loader("lazyBayesianRules")`). The new mechanism is preferred as it provides explicit information about the external package dependencies.)

(Other function affected are `DBScan`, `MultiBoostAB`, `Tertius`, and `XMeans`, for which the corresponding Java classes are now provided by Weka packages **optics_dbScan**, **multiBoostAB**, **tertius**, and **XMeans**, respectively.)

2 Persistence

A typical R work flow is fitting models and them saving them for later re-use using `save()`. It then comes as an unpleasant surprise that when the models were obtained using interfaces to Weka learners, restoring via `load()` gives "nothing". For example,

```
> m1 <- J48(Species ~ ., data = iris)
> writeLines(rJava::.jstrVal(m1$classifier))

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
|   |   |   Petal.Width <= 1.5: virginica (3.0)
|   |   |   Petal.Width > 1.5: versicolor (3.0/1.0)
|   Petal.Width > 1.7: virginica (46.0/1.0)

Number of Leaves  :         5

Size of the tree :         9

> save(m1, file = "m1.rda")
> load("m1.rda")
> rJava::.jstrVal(m1$classifier)

NULL
```

From the R side, the generated classifier is a reference to an external Java object. As such objects do not persist across sessions, they will be restored as 'null' references. Fortunately, **rJava** has added a `.jcache()` mechanism providing an R-side cache of such objects in serialized form,

which is attached to the object and hence saved when the Java object is saved, and can be restored via **rJava** mechanisms for unserializing Java references if they are 'null' references and have a cache attached. One most be cautious when creating such persistent references, though; see `?.jcache` for more information.

In our "simple" case, we can simply do

```
> m1 <- J48(Species ~ ., data = iris)
> rJava::.jcache(m1$classifier)
> save(m1, file = "m1.rda")
> load("m1.rda")
> writeLines(rJava::.jstrVal(m1$classifier))

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
|   |   |   Petal.Width <= 1.5: virginica (3.0)
|   |   |   Petal.Width > 1.5: versicolor (3.0/1.0)
|   Petal.Width > 1.7: virginica (46.0/1.0)

Number of Leaves  :         5

Size of the tree :         9
```

to achieve the desired persistence (note that the R reference object must directly be cached, not an R object containing it).

3 Interfacing Weka Graphics

RWeka currently provides no interfaces to Weka's visualization and GUI functionality: after all, its main purpose is use Weka's functionality in the usual "R look and feel". In principle, creating such interfaces is not too hard: for example, a simple interface to Weka' graph visualizer could be obtained as

```
> graphVisualizer <-
+ function(file, width = 400, height = 400,
+          title = substitute(file), ...)
+ {
+     ## Build the graph visualizer
+     visualizer <- .jnew("weka/gui/graphvisualizer/GraphVisualizer")
+     reader <- .jnew("java/io/FileReader", file)
+     .jcall(visualizer, "V", "readDOT",
+           .jcast(reader, "java/io/Reader"))
+     .jcall(visualizer, "V", "layoutGraph")
+     ## and put it into a frame.
+     frame <- .jnew("javax/swing/JFrame",
+                    paste("graphVisualizer:", title))
+     container <- .jcall(frame, "Ljava/awt/Container;", "getContentPane")
+     .jcall(container, "Ljava/awt/Component;", "add",
+            .jcast(visualizer, "java/awt/Component"))
```

```
+     .jcall(frame, "V", "setSize", as.integer(width), as.integer(height))
+     .jcall(frame, "V", "setVisible", TRUE)
+ }
```

and then used via

```
> write_to_dot(m1, "m1.dot")
> graphVisualizer("m1.dot")
```

(Currently, this fails to find the menu icon images.) Obviously, one could wrap this into plot methods for classification trees obtained via Weka's tree learners. But this would result in an R graphics window actually no longer controllable by R, which we find rather confusing. We are not aware of R graphics devices which can be used as canvas for capturing Java graphics.

Similar considerations apply for interfacing other Weka GUI functionality (such as its ARFF viewer).

4 Controlling Weka Options

The available options for the interfaced Weka classes can be queried using `WOW()`, and specified using the `control` argument to the interface functions, typically using `Weka_control()`, for which interface function arguments are replaced by their corresponding Weka Java class name, and built-in interfaces can provide additional convenience control handlers.

For example, many Weka meta learners need to distinguish options for themselves from options to be passed to the base learner, and use a special '`--`' option to separate the two sets of options. As an illustration consider the specification of J4.8 base learners with minimal leaf size of 30 in adaptive boosting. The control sequence that needs to be sent to Weka's `AdaBoostM1` classifier is

```
> c("-W", "weka.classifiers.trees.J48", "--", "-M", 30)
```

In **RWeka**, this can be passed to classifiers directly or generated more conveniently using

```
> Weka_control(W = J48, "--", M = 30)
```

where `J48()` is the registered R interface to `weka.classifiers.trees.J48`. Hence, the following calls yield the same output:

```
> myAB <- make_Weka_classifier("weka/classifiers/meta/AdaBoostM1")
> myAB(Species ~ ., data = iris,
+      control = c("-W", "weka.classifiers.trees.J48", "--", "-M", 30))
> myAB(Species ~ ., data = iris,
+      control = Weka_control(W = J48, "--", M = 30))
```

As an additional convenience the '`--`' in `Weka_control()` can be omitted in **RWeka**'s built-in meta-learner interfaces because these apply some additional internal magic to Weka control lists. Thus, the following calls yield the same output as above:

```
> AdaBoostM1(Species ~ ., data = iris,
+            control = Weka_control(W = list(J48, "--", M = 30)))
> AdaBoostM1(Species ~ ., data = iris,
+            control = Weka_control(W = list(J48, M = 30)))
```

The latter example is also used on the `AdaBoostM1()` manual page. See also the help page for `SMO()` for another example of magic performed by built-in interfaces, and the help page for `XMeans()` for another example of a low level control specification.

References

K. Hornik, C. Buchta, and A. Zeileis. Open-source machine learning: R meets Weka. *Computational Statistics*, 24(2):225–232, 2009. doi: 10.1007/s00180-008-0119-7.

S. Urbanek. *rJava: Low-Level R to Java Interface*, 2016. URL `https://CRAN.R-project.org/package=rJava`. R package version 0.9-8.

I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.