MLInterfaces 2.0 – a new design

VJ Carey

October 17, 2016

1 Introduction

MLearn, the workhorse method of MLInterfaces, has been streamlined to support simpler development.

In 1.*, MLearn included a substantial switch statement, and the external learning function was identified by a string. Many massage tasks were wrapped up in switch case elements devoted to each method. MLearn returned instances of MLOutput, but these had complicated subclasses.

MLearn now takes a signature `c("formula", "data.frame", "learnerSchema", "numeric")`, with the expectation that extra parameters captured in ... go to the fitting function. The complexity of dealing with expectations and return values of different machine learning functions is handled primarily by the learnerSchema instances. The basic realizations are that

- most learning functions use the formula/data idiom, and additional parameters can go in ...
- the problem of converting from the function's output structures (typically lists, but sometimes also objects with attributes) to the uniform structure delived by MLearn should be handled as generically as possible, but specialization will typically be needed
- the conversion process can be handled in most cases using only the native R object returned by the learning function, the data, and the training index set.
- some functions, like knn, are so idiosyncratic (lacking formula interface or predict method) that special software is needed to adapt MLearn to work with them

Thus we have defined a learnerSchema class,

```
> library(MLInterfaces)
> getClass("learnerSchema")
```

```
Class "learnerSchema" [package "MLInterfaces"]

Slots:

Name:  packageName   mlFunName   converter   predicter
Class:   character   character    function    function
```

along with a constructor used to define a family of schema objects that help MLearn carry out specific tasks of learning.

2 Some examples

We define interface schema instances with suffix "I".

randomForest has a simple converter:

```
> randomForestI@converter

function (obj, data, trainInd)
{
    teData = data[-trainInd, ]
    trData = data[trainInd, ]
    tepr = predict(obj, teData, type = "response")
    tesco = predict(obj, teData, type = "prob")
    trpr = predict(obj, trData, type = "response")
    trsco = predict(obj, trData, type = "prob")
    names(tepr) = rownames(teData)
    names(trpr) = rownames(trData)
    new("classifierOutput", testPredictions = factor(tepr), testScores = tesco,
        trainPredictions = factor(trpr), trainScores = trsco,
        RObject = obj)
}
<environment: namespace:MLInterfaces>
```

The job of the converter is to populate as much as the classifierOutput instance as possible. For something like nnet, we can do more:

```
> nnetI@converter

function (obj, data, trainInd)
{
    teData = data[-trainInd, ]
    trData = data[trainInd, ]
    tepr = predict(obj, teData, type = "class")
```

```
    trpr = predict(obj, trData, type = "class")
    names(tepr) = rownames(teData)
    names(trpr) = rownames(trData)
    new("classifierOutput", testPredictions = factor(tepr), testScores = predict(obj,
        teData), trainScores = predict(obj, trData), trainPredictions = factor(trpr),
        RObject = obj)
}
<environment: namespace:MLInterfaces>
```

We can get posterior class probabilities.

To obtain the predictions necessary for confusionMatrix computation, we may need the converter to know about parameters used in the fit. Here, closures are used.

```
> knnI(k=3, l=2)@converter

function (obj, data, trainInd)
{
    kpn = names(obj$traindat)
    teData = data[-trainInd, kpn]
    trData = data[trainInd, kpn]
    tepr = predict(obj, teData, k, l)
    trpr = predict(obj, trData, k, l)
    names(tepr) = rownames(teData)
    names(trpr) = rownames(trData)
    new("classifierOutput", testPredictions = factor(tepr), testScores = attr(tepr,
        "prob"), trainPredictions = factor(trpr), trainScores = attr(trpr,
        "prob"), RObject = obj)
}
<environment: 0x000000001426dd08>
```

So we can have the following calls:

```
> library(MASS)
> data(crabs)
> kp = sample(1:200, size=120)
> rf1 = MLearn(sp~CL+RW, data=crabs, randomForestI, kp, ntree=100)
> rf1

MLInterfaces classification output container
The call was:
MLearn(formula = sp ~ CL + RW, data = crabs, .method = randomForestI,
    trainInd = kp, ntree = 100)
Predicted outcome distribution for test set:
```

```
 B  O
41 39
Summary of scores on test set (use testScores() method for details):
      B       O
0.47625 0.52375

> RObject(rf1)

Call:
 randomForest(formula = formula, data = trdata, ntree = 100)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 1

        OOB estimate of  error rate: 40%
Confusion matrix:
   B  O class.error
B 35 23   0.3965517
O 25 37   0.4032258

> knn1 = MLearn(sp~CL+RW, data=crabs, knnI(k=3,l=2), kp)
> knn1

MLInterfaces classification output container
The call was:
MLearn(formula = sp ~ CL + RW, data = crabs, .method = knnI(k = 3,
    l = 2), trainInd = kp)
Predicted outcome distribution for test set:

 B  O
39 41
Summary of scores on test set (use testScores() method for details):
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.6667  0.6667  0.6667  0.7948  1.0000  1.0000
```

3 Making new interfaces

3.1 A simple example: ada

The `ada` method of the *ada* package has a formula interface and a predict method. We can create a learnerSchema on the fly, and then use it:

```
> adaI = makeLearnerSchema("ada", "ada", standardMLIConverter )
> arun = MLearn(sp~CL+RW, data=crabs, adaI, kp )
> confuMat(arun)

     predicted
given  B  O
    B 29 13
    O 17 21

> RObject(arun)

Call:
ada(formula, data = trdata)

Loss: exponential Method: discrete   Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
True value  B  O
         B 49  9
         O 18 44

Train Error: 0.225

Out-Of-Bag Error:  0.233  iteration= 11

Additional Estimates of number of iterations:

train.err1 train.kap1
         7         48
```

What is the standardMLIConverter?

```
> standardMLIConverter

function (obj, data, trainInd)
{
    teData = data[-trainInd, ]
    trData = data[trainInd, ]
    tepr = predict(obj, teData)
    trpr = predict(obj, trData)
    names(tepr) = rownames(teData)
    names(trpr) = rownames(trData)
```

```
    new("classifierOutput", testPredictions = factor(tepr), trainPredictions = factor(t
        RObject = obj)
}
<environment: namespace:MLInterfaces>
```

3.2 Dealing with gbm

The *gbm* package workhorse fitter is `gbm`. The formula input must have a numeric response, and the predict method only returns a numeric vector. There is also no namespace. We introduced a gbm2 function

> gbm2

```
function (formula, data, ...)
{
    requireNamespace("gbm")
    mf = model.frame(formula, data)
    resp = model.response(mf)
    if (!is(resp, "factor"))
        stop("dependent variable must be a factor in MLearn")
    if (length(levels(resp)) != 2)
        stop("dependent variable must have two levels")
    nresp = as.numeric(resp == levels(resp)[2])
    fwn = formula
    fwn[[2]] = as.name("nresp")
    newf = as.formula(fwn)
    data$nresp = nresp
    ans = gbm(newf, data = data, ...)
    class(ans) = "gbm2"
    ans
}
<environment: namespace:MLInterfaces>
```

that requires a two-level factor response and recodes for use by gbm. It also returns an S3 object of newly defined class gbm2, which only returns a factor. At this stage, we could use a standard interface, but the prediction values will be unpleasant to work with. Furthermore the predict method requires specification of `n.trees`. So we pass a parameter `n.trees.pred`.

> BgbmI

```
function (n.trees.pred = 1000, thresh = 0.5)
{
    makeLearnerSchema("MLInterfaces", "gbm2", MLIConverter.Bgbm(n.trees.pred,
```

```
        thresh))
}
<environment: namespace:MLInterfaces>

> set.seed(1234)
> gbrun = MLearn(sp~CL+RW+FL+CW+BD, data=crabs, BgbmI(n.trees.pred=25000,thresh=.5),
+    kp, n.trees=25000,
+    distribution="bernoulli", verbose=FALSE )
> gbrun

MLInterfaces classification output container
The call was:
MLearn(formula = sp ~ CL + RW + FL + CW + BD, data = crabs, .method = BgbmI(n.trees.pre
    thresh = 0.5), trainInd = kp, n.trees = 25000, distribution = "bernoulli",
    verbose = FALSE)
Predicted outcome distribution for test set:

FALSE  TRUE
   40    40
Summary of scores on test set (use testScores() method for details):
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-4.1060 -1.1860  0.4799  0.1619  1.9730  3.5310

> confuMat(gbrun)

     predicted
given FALSE TRUE
    B    33    9
    O     7   31

> summary(testScores(gbrun))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-4.1060 -1.1860  0.4799  0.1619  1.9730  3.5310
```

3.3 A harder example: rda

The *rda* package includes the rda function for regularized discriminant analysis, and rda.cv for aiding in tuning parameter selection. No formula interface is supported, and the predict method delivers arrays organized according to the tuning parameter space search. Bridge work needs to be done to support the use of MLearn with this methodology.

First we create a wrapper that uses formula-data for the rda.cv function:

```
> rdaCV = function( formula, data, ... ) {
+  passed = list(...)
+  if ("genelist" %in% names(passed)) stop("please don't supply genelist parameter.")
+  # data input to rda needs to be GxN
+  x = model.matrix(formula, data)
+  if ("(Intercept)" %in% colnames(x))
+    x = x[, -which(colnames(x) %in% "(Intercept)")]
+  x = t(x)
+  mf = model.frame(formula, data)
+  resp = as.numeric(factor(model.response(mf)))
+  run1 = rda( x, resp, ... )
+  rda.cv( run1, x, resp )
+ }
```

We also create a wrapper for a non-cross-validated call, where alpha and delta parameters are specified:

```
> rdaFixed = function( formula, data, alpha, delta, ... ) {
+  passed = list(...)
+  if ("genelist" %in% names(passed)) stop("please don't supply genelist parameter.")
+  # data input to rda needs to be GxN
+  x = model.matrix(formula, data)
+  if ("(Intercept)" %in% colnames(x))
+    x = x[, -which(colnames(x) %in% "(Intercept)")]
+  x = t(x)
+  featureNames = rownames(x)
+  mf = model.frame(formula, data)
+  resp = as.numeric(resp.fac <- factor(model.response(mf)))
+  finalFit=rda( x, resp, genelist=TRUE, alpha=alpha, delta=delta, ... )
+  list(finalFit=finalFit, x=x, resp.num=resp, resp.fac=resp.fac, featureNames=featur
+     keptFeatures=featureNames[ which(apply(finalFit$gene.list,3,function(x)x)==1) ]
+ }
```

Now our real interface is defined. It runs the rdaCV wrapper to choose tuning parameters, then passes these to rdaFixed, and defines an S3 object (for uniformity, to be like most learning packages that we work with.) We also define a predict method for that object. This has to deal with the fact that rda does not use factors.

```
> rdacvML = function(formula, data, ...) {
+  run1 = rdaCV( formula, data, ... )
+  perf.1se = cverrs(run1)$one.se.pos
+  del2keep = which.max(perf.1se[,2])
+  parms2keep = perf.1se[del2keep,]
+  alp = run1$alpha[parms2keep[1]]
```

```
+  del = run1$delta[parms2keep[2]]
+  fit = rdaFixed( formula, data, alpha=alp, delta=del, ... )
+  class(fit) = "rdacvML"
+  attr(fit, "xvalAns") = run1
+  fit
+ }
> predict.rdacvML = function(object, newdata, ...) {
+  newd = data.matrix(newdata)
+  fnames = rownames(object$x)
+  newd = newd[, fnames]
+  inds = predict(object$finalFit, object$x, object$resp.num, xnew=t(newd))
+  factor(levels(object$resp.fac)[inds])
+ }
> print.rdacvML = function(x, ...) {
+  cat("rdacvML S3 instance. components:\n")
+  print(names(x))
+  cat("---\n")
+  cat("elements of finalFit:\n")
+  print(names(x$finalFit))
+  cat("---\n")
+  cat("the rda.cv result is in the xvalAns attribute of the main object.\n")
+ }
```

Finally, the converter can take the standard form; look at rdacvI.

4 Additional features

The xvalSpec class allows us to specify types of cross-validation, and to control carefully how partitions are formed. More details are provided in the MLprac2_2 vignette.

5 The MLearn approach to clustering and other forms of unsupervised learning

A learner schema for a clustering method needs to specify clearly the feature distance measure. We will experiment here. Our main requirements are

- ExpressionSets are the basic input objects
- The typical formula interface would be `~ .` but one can imagine cases where a factor from phenoData is specified as a 'response' to color items, and this will be allowed

- a clusteringOutput class will need to be defined to contain the results, and it will propagate the result object from the native learning method.