MLInterfaces: towards uniform behavior of machine learning tools in R

VJ Carey, J Mar, R Gentleman

November 1, 2022

1 Introduction

We define machine learning methods as data based algorithms for prediction. Given data D, a generic machine learning procedure MLP produces a function ML = MLP(D). For data D' with structure comparable to D, ML(D') is a set of predictions about elements of D'.

To be slightly more precise, a dataset D is a set of records. Each record has the same structure, consisting of a set of features (predictors) and one or more predictands (classes or responses of interest, to be predicted). MLP uses features, predictands, and tuning parameter settings to construct the function ML. ML is to be a function from features only to predictands.

There are many packages and functions in R that provide machine learning procedures. They conform to the abstract setup described above, but with great diversity in the details of implementation and use. The input requirements and the output objects differ from procedure to procedure.

Our objective in *MLInterfaces* is to simplify the use and evaluation of machine learning methods by providing specifications and implementations for a uniform interface. (The `tune` procedures in *e1071* also pursue more uniform interface to machine learning procedures.) At present, we want to simplify use of machine learning with microarray data, assumed to take the form of `ExpressionSet`s. The present implementation addresses the following concerns:

- simplify the selection of the predictand from `ExpressionSet` structure;
- simplify (in fact, require) decomposition of input data into training and test set, with output emphasizing test set results;
- provide a uniform output structure.

The output structures currently supported are subclasses of a general class *MLOutput*, described in Section ?? below.

To give a flavor of the current implementation, we perform a few runs with different machine learning tools. We will use 60 genes drawn arbitrarily from Golub's data.

```
> library(MLInterfaces)
> library(golubEsets)
> library(genefilter)

> data(Golub_Merge)
> smallG <- Golub_Merge[200:259,]
> smallG

ExpressionSet (storageMode: lockedEnvironment)
assayData: 60 features, 72 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: 39 40 ... 33 (72 total)
  varLabels: Samples ALL.AML ... Source (11 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
  pubMedIds: 10521349
Annotation: hu6800
```

Here is how k-nearest neighbors is used to get predictions of ALL status, using the first 40 records as the training set:

```
> #krun <- knnB( smallG, "ALL.AML", trainInd=1:40 )
> krun = MLearn(ALL.AML~., smallG, knnI(k=1), 1:40)

[1] "ALL.AML"

> krun

MLInterfaces classification output container
The call was:
MLearn(formula = ALL.AML ~ ., data = smallG, .method = knnI(k = 1),
    trainInd = 1:40)
Predicted outcome distribution for test set:

ALL AML
 22  10
Summary of scores on test set (use testScores() method for details):
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       1       1       1       1       1
```

The `confuMat` method computes the confusion matrix resulting from applying the trained model to the reserved test data:

```
> confuMat(krun)
```

```
     predicted
given ALL AML
  ALL  18   3
  AML   4   7
```

Additional parameters can be supplied as accepted by the target procedure in package *class*. To use a neural net in the same context (with fewer genes to simplify the summary below)

```
> set.seed(1234)
> #nns <- nnetB( smallG[1:10,], "ALL.AML", trainInd=1:40, size=2, decay=.01, maxit=250
> nns <- MLearn( ALL.AML~., smallG[1:10,], nnetI, trainInd=1:40, size=2, decay=.01, max
```

```
[1] "ALL.AML"
# weights:  25
initial  value 27.327352
iter  10 value 25.018790
iter  20 value 21.799662
iter  30 value 19.267699
iter  40 value 14.859899
iter  50 value 10.836288
iter  60 value 8.196922
iter  70 value 8.182311
iter  80 value 7.381528
iter  90 value 7.291806
iter 100 value 7.204848
iter 110 value 7.185043
iter 120 value 7.159077
iter 130 value 6.803327
iter 140 value 6.779275
iter 150 value 6.778553
iter 160 value 6.778405
iter 170 value 6.778377
iter 180 value 6.778356
iter 190 value 6.778342
iter 190 value 6.778342
iter 190 value 6.778342
final  value 6.778342
converged
```

```
> nns

MLInterfaces classification output container
The call was:
MLearn(formula = ALL.AML ~ ., data = smallG[1:10, ], .method = nnetI,
    trainInd = 1:40, size = 2, decay = 0.01, maxit = 250)
Predicted outcome distribution for test set:

ALL AML
 24   8
Summary of scores on test set (use testScores() method for details):
[1] 0.2690627

> confuMat(nns)

     predicted
given ALL AML
  ALL  18   3
  AML   6   5
```

2 Usage

The basic call sequence for supervised learning for `ExpressionSet`s is

```
MLearn(formula, data, learnerSchema, trainInd, ...)
```

The parameter `formula` is a standard R formula, with `y~xz+` indicating that `x and \verbz+` are predictors of response `y`. If `data` is a data.frame instance, then the formula has the usual interpretation for R. If `data` is an `ExpressionSet` instance, then it is assumed that the dependent variable is present in the pData component of phenoData, and the variables on the RHS are found in the exprs component of assayData. If `.` is used on the RHS, then all features in the exprs component are used as predictors. The `learnerSchema` parameter is bound by instances of the learnerSchema class. Many examples are provided with MLInterfaces, see the page from help(MLearn) for a complete list. Parameter `trainInd` is a numeric sequence isolating the samples to be used for training; it may also be bound by an instance of xvalSpec to define a cross-validation of a learning process (see section **??**).

3 Classes

For input to MLearn, to define the procedure to be used, two major classes are defined: learnerSchema, and xvalSpec.

```
> getClass("learnerSchema")

Class "learnerSchema" [package "MLInterfaces"]

Slots:

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

> getClass("xvalSpec")

Class "xvalSpec" [package "MLInterfaces"]

Slots:

Name:           type         niter partitionFunc         fsFun
Class:     character       numeric      function      function
```

For output, we have only the classifierOutput class:

```
> getClass("classifierOutput")

Class "classifierOutput" [package "MLInterfaces"]

Slots:

Name:          trainInd     testOutcomes  testPredictions       testScores
Class:          numeric           factor           factor              ANY

Name:     trainOutcomes trainPredictions      trainScores        fsHistory
Class:           factor           factor              ANY             list

Name:           RObject             call       embeddedCV    learnerSchema
Class:              ANY             call          logical    learnerSchema
```

4 Cross-validation

Instances of the xvalSpec class are bound to the trainInd parameter of MLearn to perform cross-validation. The constructor `xvalSpec` can be used in line. It has parameters type (only relevant to select "LOO", for leave-one out), niter (number of partitions to use), partitionFunc (function that returns indices of members of partitions), fsFunc (function that performs feature selection and returns a formula with selected features on right-hand side).

The partitionFunc must take parameters data, clab, iternum. data is the usual data frame to be supplied to the learner function. clab must be the name of a column in data. Values of the variable in that column are balanced across cross-validation partitions. iternum is used to select the partition elements as we iterate through cross validation.

- straight leave-one-out (LOO) – note the group parameter must be integer; it is irrelevant for the LOO method.

```
> library(golubEsets)
> data(Golub_Merge)
> smallG <- Golub_Merge[200:250,]
> lk1 <- MLearn(ALL.AML~., smallG, knnI(k=1,l=0), xvalSpec("LOO"))

[1] "ALL.AML"

> confuMat(lk1)

     predicted
given ALL AML
  ALL  37  10
  AML  10  15
```

- Now do a random 8-fold cross-validation.

```
> ranpart = function(K, data) {
+  N = nrow(data)
+  cu = as.numeric(cut(1:N, K))
+  sample(cu, size=N, replace=FALSE)
+ }
> ranPartition = function(K) function(data, clab, iternum) {
+  p = ranpart(K, data)
+  which(p != iternum) # to retain training fraction
+ }
> lkran <- MLearn(ALL.AML~., smallG, knnI(k=1,l=0), xvalSpec("LOG", 8, partitionFu

[1] "ALL.AML"

> confuMat(lkran)

     predicted
given ALL AML
  ALL  26   7
  AML   7  10
```

- Now do an 8-fold cross-validation with approximate balance among groups with respect to frequency of ALL and AML. The utility function balKfold.xvspec helps for this.

```
> lk3 <- MLearn(ALL.AML~., smallG, knnI(k=1,l=0), xvalSpec("LOG", 8, partitionFunc

[1] "ALL.AML"

> confuMat(lk3)

     predicted
given ALL AML
  ALL  39   8
  AML   8  17
```

5 Cross validation with feature selection

Stephen Henderson of UC London supplied infrastructure to allow embedding of feature selection in the cross-validation process. These have been wrapped in fs.* closures that can be passed in xvalSpec:

```
> data(iris)
> iris2 = iris[ iris$Species %in% levels(iris$Species)[1:2], ]
> iris2$Species = factor(iris2$Species) # drop unused levels
> x1 = MLearn(Species~., iris2, ldaI, xvalSpec("LOG", 3,
+    balKfold.xvspec(3), fs.absT(3)))
> fsHistory(x1)

[[1]]
[1] "Petal.Length" "Petal.Width"  "Sepal.Length"

[[2]]
[1] "Petal.Length" "Petal.Width"  "Sepal.Length"

[[3]]
[1] "Petal.Length" "Petal.Width"  "Sepal.Width"
```

6 A sketch of a 'doubt' computation

The `nnet` function returns a structure encoding predicted probabilities of class occupancy. We will use this to enrich the `MLearn/nnetI` output to include a "doubt" outcome. As written this code will handle a two-class outcome; additional structure emerges with more than two classes and some changes will be needed for such cases.

First we obtain the predicted probabilities (for the test set) and round these for display purposes.

```
> predProb <- round(testScores(nns),3)
```

We save the true labels and the predicted labels.

```
> truth <- as.character(smallG$ALL.AML[-c(1:40)])
> simpPred <- as.character(testPredictions(nns))
```

We create a closure that allows boundaries of class probabilities to be specified for assertion of "doubt":

```
> douClo <- function(pprob)  function(lo,hi) pprob>lo & pprob<hi
```

Evaluate the closure on the predicted probabilities, yielding a function of two arguments (lo, hi).

```
> smallDou <- douClo(predProb)
```

Now replace the labels for those predictions that are very close to .5.

```
> douPred <- simpPred
> douPred[smallDou(.35,.65)] <- "doubt"
```

The resulting modified predictions are in the fourth column:

```
> mm <- cbind(predProb,truth,simpPred,douPred)
> mm
```

```
           truth simpPred douPred
7  "0.643" "ALL" "AML"    "doubt"
8  "0.045" "ALL" "ALL"    "ALL"
9  "0.045" "ALL" "ALL"    "ALL"
10 "0.045" "ALL" "ALL"    "ALL"
11 "0.045" "ALL" "ALL"    "ALL"
12 "0.94"  "ALL" "AML"    "AML"
13 "0.045" "ALL" "ALL"    "ALL"
14 "0.045" "ALL" "ALL"    "ALL"
15 "0.045" "ALL" "ALL"    "ALL"
16 "0.045" "ALL" "ALL"    "ALL"
17 "0.999" "ALL" "AML"    "AML"
18 "0.045" "ALL" "ALL"    "ALL"
19 "0.045" "ALL" "ALL"    "ALL"
20 "0.045" "ALL" "ALL"    "ALL"
21 "0.045" "ALL" "ALL"    "ALL"
```

```
22 "0.045" "ALL" "ALL"    "ALL"
23 "0.045" "ALL" "ALL"    "ALL"
24 "0.045" "ALL" "ALL"    "ALL"
25 "0.045" "ALL" "ALL"    "ALL"
26 "0.045" "ALL" "ALL"    "ALL"
27 "0.045" "ALL" "ALL"    "ALL"
34 "0.999" "AML" "AML"    "AML"
35 "0.999" "AML" "AML"    "AML"
36 "0.045" "AML" "ALL"    "ALL"
37 "0.045" "AML" "ALL"    "ALL"
38 "0.045" "AML" "ALL"    "ALL"
28 "0.045" "AML" "ALL"    "ALL"
29 "0.045" "AML" "ALL"    "ALL"
30 "0.045" "AML" "ALL"    "ALL"
31 "0.999" "AML" "AML"    "AML"
32 "0.999" "AML" "AML"    "AML"
33 "0.947" "AML" "AML"    "AML"
```

```
> table(mm[,"truth"], mm[,"simpPred"])

      ALL AML
  ALL  18   3
  AML   6   5
```

```
> table(mm[,"truth"], mm[,"douPred"])

      ALL AML doubt
  ALL  18   2     1
  AML   6   5     0
```