Using `xval` and clusters of computers for cross-validation

MT Morgan (mtmorgan@fhcrc.org)

2 December, 2005

1 Introduction

Cross-validation and other methods like the bootstrap can be 'embarrassingly parallel'. Embarrassingly parallel means that the same calculation is repeated independently for different data sets. One way to decrease the execution time of embarrassingly parallel computations is to use a cluster of computers. Each node in the cluster performs part of the computation, dividing the execution time by 1 over the number of nodes in the cluster.

This document illustrates how clusters can perform cross-validation calculations, using the `xval` method from the *MLInterfaces* package. The document starts with a brief overview of what a cluster computation might look like, with some essential details swept 'under the rug'; this is the 'end-user' experience. Delving deeper into the steps required for this experience is the 'R programmer' experience. This section outlines the implementation required (not provided in *MLInterfaces*!) for clustered computing, the likely performance gains, and directions for developing more effective parallelization strategies. The section concludes with a summary of steps taken to make `xval` more amenable to clustered evaluation; these may be useful guides for exposing other functions to parallelization.

2 Overview – the 'end user' experience

A non-clustered cross-validation might involve the following `R` commands, taken from the `xval` documentation examples:

```
> library(MLInterfaces)
> library(golubEsets)
> data(Golub_Merge)
> smallG <- Golub_Merge[200:250,]
```

The first three statements load the *MLInterfaces* package and a data set. The next line extracts a portion of the data set for use in the example. We then perform a "leave one out" (LOO) cross-validation, and summarize the results in a table:

```
> lk1 <- MLearn(ALL.AML~., smallG, knnI(k=1), xvalSpec("LOO"))

[1] "ALL.AML"

> confuMat(lk1)

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

A clustered version of the same calculation requires that the user have a computer cluster available for `R` to use. The cluster has software libraries installed to allow communication between cluster nodes. Common software packages include `MPI` and `PVM`. `R` also requires packages that allow communication between `R` and the cluster software. Available packages include *Rmpi* and *rpvm*; usually one installs the package corresponding to the cluster library available on the system. Both *Rmpi* and *rpvm* provide a fairly 'low-level' interface to the clusters. A better starting point adds the package *snow* as another layer on top of *Rmpi* or *pvm*. *snow* makes it easy to start and stop clusters, and provides a uniform interface to key cluster functions.

Assuming a functioning cluster with appropriate `R` packages, and with one caveat, the clustered cross-validation might start as before...

```
> library(MLInterfaces)
> library(golubEsets)
> smallG <- Golub_Merge[200:250,]
```

Then load *snow*, start the cluster (e.g., with 8 nodes), and load *MLInterfaces* on each node...

```
> library(snow)
> cl <- makeCluster(8, "MPI")
> clusterEvalQ(cl, library(MLInterfaces))
```

Finally, perform the calculation across the cluster... FOR THIS TO WORK WE WILL HAVE TO EXTEND xvalSpec TO DEAL WITH makeCluster OBJECTS.

```
> lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod="LOO",
+             group=as.integer(0), cluster = cl)
> table(lk1,smallG$ALL.AML)
```

Notice that the only change to the actual `xval` call is the inclusion of the `cluster = cl` argument. Easy!

3 Caveats – the 'R programmer' experience

NOTE THAT THIS MATERIAL HAS TO BE UPDATED TO DEAL WITH THE MLearn/learnerSchema ARCHITECTURE THAT PREDATED THIS MATERIAL.

Now the caveat. Someone has to tell `xval` how to do the calculation in clustered mode. This is the job of the generic function `xvalLoop`. We'll build up to a workable solution in this section.

Take a peak at the code inside the `xval` method (e.g., typing `getMethods(xval)` at the `R` command prompt). You will see `xvalLoop` appearing near the top, in a line reading `xvalLoop <- xvalLoop(cluster)`. The function `xvalLoop` is a generic function, and it is called with the argument `cluster`. Here's the definition of the default method of `xvalLoop` function:

```
> setMethod("xvalLoop",
+           signature( cluster = "ANY" ),
+           function( cluster, ... ) lapply )
```

This code is executed when `cluster = NULL` (the default, when no `cluster` argument is provided). By default, then, the `xvalLoop` function returns the `lapply` function. The value of the *variable* `xvalLoop` (i.e., on the left of `xvalLoop <- xvalLoop(cluster)` is then `lapply`. A bit confusing, but hopefully not too confusing.

Look further into the `xval` code. You'll see `xvalLoop` appearing again, toward the base of the method, as `out <- xvalLoop( 1:n, xvalidator, ... )`. Since the value of `xvalLoop` is `lapply`, this is the same as `out <- lapply( 1:n, xvalidator, ... )`.

We would like our clustered version of `xval` to behave differently, specifically to use an `lapply`-like function that takes advantage of the computer cluster. We do this by writing a method of `xvalLoop` that is specific to our type of cluster. Suppose, as in the example above, we have an MPI cluster created with *snow*. Some reading of the *snow* documentation leads to the `parLapply` function. This behaves just like `lapply`, but distributes the tasks over cluster nodes. In an ideal world, we would be able to write

```
> setOldClass( "spawnedMPIcluster" )
> setMethod("xvalLoop",
+           signature( cluster = "spawnedMPIcluster" ),
+           function( cluster, ... ) parLapply )
```

Suppose we provided the argument `cluster = cl` to the `xval` function. After `xvalLoop <- xvalLoop(cluster)`, the value of the variable `xvalLoop` is `parLapply`. Later in `xval`, we would execute the equivalent of `out <- parLapply( 1:n, xvalidator, ... )`. If only it were that easy!

There are two problems with the approach developed so far. The first is that the arguments of `parLapply` are different from those of `lapply`. Specifically, the first argument of `parLapply` is the cluster.

Here is one solution to this problem. Create a 'wrapper' for `parLapply` inside the `xvalLoop` method for `spawnedMPIcluster` that hides the cluster argument. Return the wrapper function, rather than `parLapply`:

```
> setOldClass( "spawnedMPIcluster" )
> setMethod("xvalLoop",
+ signature( cluster = "spawnedMPIcluster" ),
+ function( cluster, ... ) {
+     relapply <- function(X, FUN, ...) {
+         parLapply( cluster, X, FUN, ... )
+     }
+     relapply
+ })
```

Back in `xval`, the value of the variable `xvalLoop` is now `relapply`. The crucial call later in `xval` will be `out <- relapply( 1:n, xvalidator, ... )`. The function `relapply` is called with the arguments `1:n`, `xvalidator`, and `...`. `relapply` then calls `parLapply` with the arguments `cluster` and the other arguments from the call to `relapply`. The details of how `cluster` gets assigned the correct value involve the lexical scoping rules of `R`, but what happens is that `cluster` takes on the value it had when the `xvalLoop` method was invoked – the `relapply` function remembers the environment in which it was created, and the environment includes the variable `cluster`.

And now the second problem. As it stands, only the computer executing `xval` knows about the various variables that have been defined, and are needed, for the cross-validation calls. We have to share these variables with all cluster nodes contributing to the cross-validation. *snow* allows us to export variables from one node to another. This is a slow process, because in *snow* a new communication channel is established for each variable. In addition, we would have to fully understand the `xval` method to know which variables needed to be exported under which circumstances.

Here is a solution that makes variable export quick while removing the need for detailed knowledge of which specific variables are required: bundle all the 'visible' variables into an environment, export the environment to each node, and unpack the environment at the receiving node. This turns out to be a good solution, because virtually all the visible variables will be needed in the cross-validation. The implementation of this solution is as follows:

```
> setMethod("xvalLoop", signature( cluster = "spawnedMPIcluster"),
+ function( cluster, ... ) {
+     clusterExportEnv <- function (cl, env = .GlobalEnv)
+     {
+         unpackEnv <- function(env) {
+             for ( name in ls(env) ) assign(name, get(name, env), .GlobalEnv )
+             NULL
+         }
+         clusterCall(cl, unpackEnv, env)
```

```
+     }
+     relapply <- function(X, FUN, ...) {
+         ## send all visible variables from the parent (i.e., xval) frame
+         clusterExportEnv( cluster, parent.frame(1) )
+         parLapply( cluster, X, FUN, ... )
+     }
+     relapply
+ })
```

The `clusterExportEnv` function is responsible for taking variables in an environment and sending them and a function to 'unpack' the environment to each node. `clusterCall` is defined in *snow*.

While tortuous in development from scratch, the end result and steps for creating new methods is straight-forward: create an `xvalLoop` method for a clustered environment that (a) exports visible variables and (b) implements and returns a parallelized `lapply`-like function.

3.1 Performance

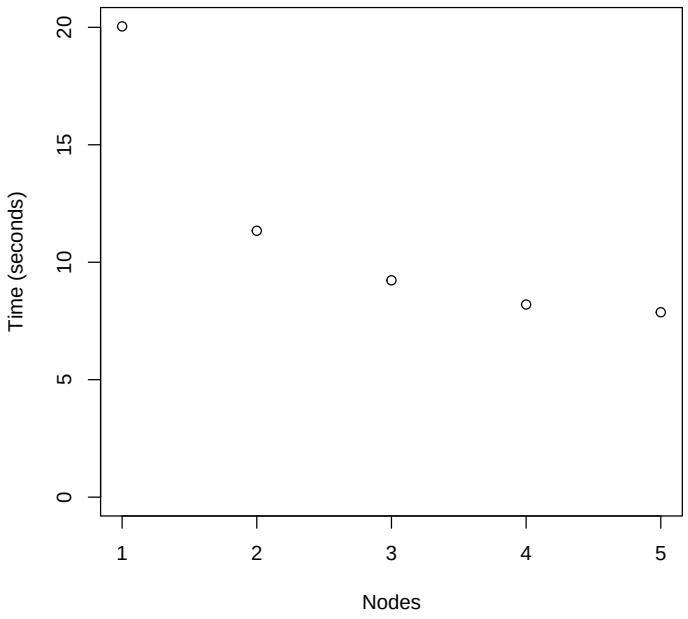

Figure 1: Execution time decreases asymptotically with number of nodes. The decreasing performance gain is primarily due to communication overhead.

After all this work, what do we get? Without immediate access to a cluster, I ran `xval` on a collection of 64-bit x86 linux computers linked through standard ethernet switches. There are a variety of users on each node, so they may not perform equivalently. The code used is

```
> harness <- function( nodes, reps, data ) {
+     if ( nodes > 1) {
+         cl <- makeCluster(nodes, "MPI")
+         clusterEvalQ(cl, library(MLInterfaces))
+     } else cl <- NULL
+     func <- function(x)
+         res <- xval(data, "ALL.AML", knnB, xvalMethod = "LOO", 0:0, cluster = cl )
+     func()                                # warm-up
+     tm <- system.time( sapply( 1:reps, func ) )[3]
+     if (nodes > 1) stopCluster(cl)
+     tm
+ }
> res <- sapply( 1:5, harness, 10, smallG )
```

Figure 1 summarizes timings from 10 replicate calculations of the cross-validation used above. The details of the results are likely to be quite system- and problem-dependent. However, a general point emerges.

A basic framework for thinking about total execution time is to recognize two components: computation, and communication. In an ideal world, computation time and node number are inversely related nodes, $t_{\text{comp}} \propto 1/n$. Communication time depends on how tasks are communicated to nodes. In *snow*, communication increases linearly with node number, $t_{\text{comm}} \propto n$. The combination of these times means that there is a node number that minimizes overall execution time. Results in Figure 1 suggest that the optimum cluster size for this problem is surprisingly small.

3.2 Improvements

This section sketches two ways to improve execution time. A goal is to develop this more fully with worked examples.

snow and `clusterCall` communication scales linearly with node number. This is because of the way *snow* implements cluster-wide calls. Specifically, the 'master' node (i.e., the node running `R` through which interactions occur) communicates separately with each slave node. Both `MPI` and `PVM` allow for more efficient models of communication, through `broadcast` operations that synchronize data across nodes in $\log n$ time.

The `broadcast` approach is likely to be important in large clusters, but in the example of the previous section the efficiency is not that great. The underlying problem is the transfer of a large volume of data; actually, the optimum node number generally *decreases* as the size of the data set increases. One solution to this is somewhat counter-intuitive, at least to me. The idea is for all nodes to independently read in data and execute all code up to the location where

`xvalLoop` is used in `xval`. The 'slave' nodes then each execute a portion of the cross-validations, while the 'master' node collates results from the slaves. At the end of `xvalLoop` the master has a complete collection of results and continues on; the slaves have an incomplete set of results, and can be terminated. The code is counter-intuitive because each node performs seemingly redundant calculations (the processing up to `xvalLoop`). Communication times should be greatly reduced, though, because each communication involves just a short set of results rather than a potentially very large data set. An assumption is that the clustered file system will cope well with near-simultaneous requests to initially read in data.

3.3 Making `xval` cluster-friendly

The implementation presented above required revision of the `xval` code. The structure of the routine is: (1) check on supplied variables; (2) preliminary set-up; (3) a loop-like call where the 'work' of the function occurs; and (4) processing of the result. Many high-level `R` routines have a similar structure. The revisions aimed to more cleanly distinguish the overall structure, and in particular to isolate and refactor part (3).

A key step in the refactoring was to change a `for` loop into an `lapply`-like function call. The reason that this is so important is that `for` loops allow side-effects to seep beyond the loop. Side-effects are much more easily avoided in a function call. Here is a simple illustration, where the 'side-effect' in the `for` loop changes the value of `x`:

```
> x <- 1:10
> for (i in 1:10) x[i] <- i**2
> x # x has been modified by the for loop

 [1]   1   4   9  16  25  36  49  64  81 100

> x <- 1:10
> res <- lapply(1:10, function(i) x[i] <- i**2)
> x # copy of x in lapply modified, not the global x

 [1]  1  2  3  4  5  6  7  8  9 10
```

Avoiding side-effects is particularly important in a cluster, because side-effects are limited to the node on which the calculation occurs. Usually this is not what is desired.

A final step in code revision is the hook, `xvalLoop`, provided for programmer access. `xvalLoop` provides a division-of-labor between the `xval` expert and the clustered computing expert. The `xval` expert needs only know that clustered computers can take advantage of `lapply`-like code. The clustered computing expert need know little about `xval`, just that it has an `lapply`-like loop to be exploited. Since `xvalLoop` is a generic function, the clustered computing expert can tailor methods to different types of clusters and even different algorithms. The end user need know little about either `xval` or clustered computing; they just provide their cluster as an argument to `xval`.

4 Conclusion

This document provides a guide to users wanting to make use of `xval` on a computer cluster. It also provides guidance for further parallelization of `xval`, and outlines how functions can be written to expose key components to parallelization. An emphasis is the division of labor between specialized function writers, clustered computer experts, and end users.