Optimizing Regularization with RcppArmadillo & bigmemory

Pete Mohanty and Robert B Shaffer

12/11/2016

The regularization parameter λ is found by Golden Search. Based on the eigenvalues and a handful of constants, *lambdasearch* defines the outer bounds of the search space and then proceeds to search for the peak of a unimodal function. Until a value for λ is found that is within in the desired (user defined) threshold, *solveforc* is called and does the heavy lifting. This process becomes extremely time consuming as sample size grows because it involves cross-multiplying an N x N matrix of eigenvectors by a version itself wherein each entry is modified by the working hypothesis for λ. On a typical laptop, *bigKRLS::bLambdaSearch* might take half an hour once N > 5,000 (assuming 16 iterations to reach the default threshold) using RcppArmadillo and a "naive" linear algebra approach along the lines of Rifkin and Lippert 2007.

This document walks through the problem in base R, explains the key features of our algorithm which is designed to boost speed and reduce memory overhead, and then outlines our particular implementation with RcppArmadillo and bigmemory. For a more general introduction to this project see our working paper, slides, and other materials available here.

The first task is to compute the vector of coefficients, $\mathbf{c}$ as a function of the eigenvectors and values, λ, and $\mathbf{y}$. Here is base R code which is similar to that found in Hainmueller and Hazlett's KRLS R package:

```
Ginv <- tcrossprod(KRLS::multdiag(X = Eigenobject$vectors, d = 1/(Eigenobject$values +
    lambda)), Eigenobject$vectors)
coeffs <- Ginv %*% y
```

The other task is computing *looloss* (leave one out error loss) as a function of $\mathbf{c}$ and the diagonal of G^{-1}.

```
Le <- crossprod(coeffs/diag(Ginv))    # looloss
```

Since the coeffs is just a vector, crossprod(coeffs/diag(Ginv)) divides each coefficient by the corresponding element of Ginv's diagonal and then computes the sum of squares. Once λ is found, *solveforc* function is called once more.

This whole process gets slows as N grows since we are computing a nuisance N $*$ N matrix, G^{-1}, at each iteration of the search. Many of these calculations cannot be avoided since there is no easy way to isolate λ. However, G^{-1} is symmetric, which can perhaps be seen more clearly from Rifkin and Lippert (2007, 5). Each element of G^{-1} can be written:

$$G^{-1}_{i,j} = \sum_{k=1}^{n} \frac{Q_{i,k} * Q_{j,k}}{\Lambda_{k,k} + \lambda}$$

Where Q stores the eigenvectors and Λ is a matrix with eigenvalues on the diagonal and zeros off diagonal.

G^{-1} is symmetric since relevant eigenvector entries are just multiplied and the denominator is independent of i and j.

```
Ginv <- tcrossprod(KRLS:::multdiag(X = Eigenobject$vectors, d = 1/(Eigenobject$values +
    lambda)), Eigenobject$vectors)
isSymmetric(Ginv)
```

```
## [1] TRUE
```

There's no need to construct G^{-1} at all provided that the entries which would go on its diagonal are stored. Here's some demo base R code with values initialized to the start of a Golden Search that shows how to skip the constuction of G^{-1} and avoid redundant calculations.

```
Ginv.test <- Ginv.diag <- c <- matrix(nrow = n, 0)

for (i in 1:n) {
    for (j in 1:i) {
        # does lower left triangle
        g <- 0
        for (k in 1:n) {
            g <- g + (Eigenobject$vectors[i, k] * Eigenobject$vectors[j, k])/(lambda +
                Eigenobject$values[k])
        }
        if (i == j) {
            Ginv.diag[i] <- g
        } else {
            c[j] <- c[j] + g * y[i]
        }
        c[i] <- c[i] + g * y[j]
    }
}
max(abs(diag(Ginv) - Ginv.diag))
```

```
## [1] 3.469447e-18
```

```
Le <- crossprod(c/diag(Ginv))
Le.test = crossprod(c/Ginv.diag)
Le - Le.test
```

```
##      [,1]
## [1,]    0
```

Adapting this to C++ proved less straightfoward than anticipated. Surprisingly, not all versions of the algorithm that took advantage of G^{-1} symmetry ran faster than the linear algebra approach despite doing half the calculations! After some investigation, we identified the step in which we copied the submatrix temp_eigen at each step of the loop as the most costly part of the algorithm. We addressed this problem by converting the matrix to a pointer to the original matrix:

```
mat temp_eigen(Eigenvectors.memptr(), N, i+1, false);
```

Pointers to submatrices cannot be operated on (e.g. transposed) without changing the original matrix, which was also problematic. We addressed this issue by transposing the entire eigenvector matrix before beginning calculations, with a second transposition after calculations were complete.

```
template <typename T>
List xBigSolveForc(Mat<T> Eigenvectors, const colvec Eigenvalues,
                   const colvec y, const double lambda){

  double Le = 0;

  int N = Eigenvectors.n_rows;

  colvec Ginv_diag(N); Ginv_diag.zeros();
  colvec coeffs(N); coeffs.zeros();

  Eigenvectors = trans(Eigenvectors);
```

```
  for(int i = 0; i < N; i++){
    colvec g(i);

    mat temp_eigen(Eigenvectors.memptr(), N, i+1, false);

    g = (Eigenvectors.col(i).t()/(Eigenvalues + lambda)) * temp_eigen;

    Ginv_diag[i] = g[i];
    coeffs(span(0,i-1)) += g * y[i];
    coeffs[i] += sum(g * y(span(0,i)));
  }

  Eigenvectors = trans(Eigenvectors);

  for(int i = 0; i < N; i++){
    Le += pow((coeffs[i]/Ginv_diag[i]), 2);
  }

  List out(2);
  out[0] = Le;
  out[1] = coeffs;

  return out;
}
```

Here is an example with at N = 1000 (on a mid 2012 MacBook Pro with 8 gigs of RAM). At this sample size, KRLS and the old bigKRLS algorithm perform comparably (likely because crossprod is well implemented in base R, simply switching to Rcpp doesn't offer speed gains like ones we've documented elsewhere with bigKRLS). Our new algorithm outperforms both.

```
dim(Eigenobject$vectors)
```

```
## [1] 1000 1000
```

```
y_baseR <- as.matrix(y)
eigen_baseR <- list()
eigen_baseR$vectors <- as.matrix(Eigenobject$vectors)
eigen_baseR$values <- Eigenobject$values

system.time(out_baseR <- KRLS::solveforc(y_baseR, Eigenobject = eigen_baseR,
    lambda = lambda, eigtrunc = NULL))
```

```
##    user  system elapsed
##   0.912   0.011   0.928
```

```
system.time(out_old <- bigKRLS::bSolveForc(Eigenobject = Eigenobject, y = y,
    lambda = lambda))
```

```
##    user  system elapsed
##   0.977   0.019   1.166
```

```
system.time(out_new <- bSolveForc_new(Eigenobject = Eigenobject, y = y, lambda = lambda))
```

```
##    user  system elapsed
##   0.538   0.002   0.542
```