Statistical Programming, Week-Long Practical

Marco Scutari

April 23, 2018

This practical sheet contains a single, assessed exercise on which you should write a report with a soft word limit at 2000 words and a hard limit at 2500 words. Also note that you should use the anonymous practicals ID (and not your real name) for the cover page of the report, and you should name the PDF file you upload to WebLearn using that same ID (e.g., "P042.pdf").

In this particular practical report it is fine to include snippets of code in the main body of the report for the purpose of illustrating and discussing them.

Exercise : Linear Regression for Correlated Data

Consider a linear regression model of the form

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon} \qquad \text{with} \qquad \boldsymbol{\varepsilon} \sim N(0, sK + tI_n) \tag{1}$$

where y is an $n \times 1$ vector; X is a $n \times p$ matrix (with p explanatory variables, assuming no intercept); $s, t \in \mathbb{R}^+$; K is an $n \times n$ covariance matrix; and I is the $n \times n$ identity matrix. This kind of model is very general and it is widely used in genetics (where individuals in the sample may be related) and environmental sciences (where measurements are correlated in time and space).

If we write a spectral decomposition of $K = U\Lambda U^T$, then

$$sK + tI_n = sU\Lambda U^T + tUU^T = U(s\Lambda + tI)U^T \tag{2}$$

which we can further simplify to $U(h\Lambda + (1-h)I)U^T$ if we set $h = s/(s+t) \in [0,1]$, leaving only one tuning parameter. Hence

$$U^Ty = U^TX\boldsymbol{\beta} + U^T\boldsymbol{\varepsilon} \qquad \text{with} \qquad U^T\boldsymbol{\varepsilon} \sim N(0, (s+t)(h\lambda_i + (1-h))) \tag{3}$$

which is a weighted regression in which observations are independent with weights $w_i = h\lambda_i + (1-h)$, $i = 1, \ldots, n$.

1. Implement an R function to estimate the model (1) via the formulation in (3) assuming h and K are known, without using `lm()`. The function should take y, X, h and K as arguments; and it should be implemented using the best practices we discussed in the lectures (checking the validity of the argument values, using comments, indentation, etc.). The function should return a vector containing the estimated values for $\boldsymbol{\beta}$, and it should be called `lmKfit()`.

2. Implement a unit test checking that `lmKfit()` gives the correct estimates for the $\boldsymbol{\beta}$. Use `lm()` to produce "known good" estimates to compare against, and *rmvnorm*() from the **mvtnorm** package to generate a random sample to run the test. The test should produce an error if the estimates are incorrect.

3. What is the computational complexity of `lmKfit()`? Double-check your results for the complexity in the sample size by simulation.

4. Implement a function `lmKhfit()` that estimates h as well as $\boldsymbol{\beta}$, and returns both in a list. (*Hint: you can use* `optimize() to do that`.)

5. Implement a parallel version of `lmKhfit()` (say, `parlmKhfit()`) that estimates h by exploring a dense grid of values over $[0.01, 0.899]$. (Values outside of this interval cause numerical problems, so for simplicity you can disallow them in optimising for h.)

6. Which is faster, `lmKhfit()` or `parlmKhfit()`?

Solution to Exercise

1. The function to fit the model in (1) is the `lmKfit()` below:

```
lmKfit = function(y, X, h, K) {

  if (!is.numeric(y) || !is.vector(y))
    stop("'y' must be a numeric vector.")
  if (!is.numeric(X) || !is.matrix(X))
    stop("'X' must be a numeric matrix.")
  if (!is.numeric(h) || length(h) > 1)
    stop("'h' must be a single number.")
  if (h > 1 || h < 0)
    stop("'h' must be a single number in [0, 1].")

  nobs = length(y)

  if (nrow(X) != nobs)
    stop("'y' and 'X' contain different numbers of observations.")
  if ((nrow(K) != ncol(K)) || (nrow(K) != nobs))
    stop("'K' must be a square matrix with ", nobs, " rows.")

  # compute the spectral decomposition of K.
  e = eigen(K)
  # rotate both X and y.
  Ut = t(e$vectors)
  yrot = Ut %*% y
  Xrot = Ut %*% X
  # compute the weights from the eigenvalues and h.
  w = 1/(h * e$values + (1 - h))

  # estimate the betas using the closed-form estimator for weighted
  # least squares.
  Xwt = t(Xrot * w)
  ywt = t(yrot * w)
  XwX = Xwt %*% Xrot
  Xwy = Xwt %*% yrot
  beta = solve(XwX) %*% Xwy

  return(as.vector(beta))

}#LMKFIT
```

2. We can generate a random sample as follows:

```
library(mvtnorm)
n = 1000
h = 0.75
X = matrix(rnorm(5 * n), nrow = n, ncol = 5)
beta = (1:5)/5
K = diag(0.5, n) + 0.5
K[1, 2:n] = K[2:n, 1] = 0.75
epsilon = t(rmvnorm(1, sigma = h * K + (1 - h) * diag(n)))
y = as.vector(X %*% beta + epsilon)
```

   Then the unit test can be implemented as:

```
estimated = lmKfit(y = y, X = X, h = h, K = K)

e = eigen(K)
Ut = t(e$vectors)
w = 1/(h * e$values + (1 - h))
reference = coef(lm(I(Ut %*% y) ~ I(Ut %*% X) - 1, weights = w))

stopifnot(all.equal(estimated, reference, check.attributes = FALSE))
```

   Note that without `check.attributes = FALSE` the test fails because `coef(lm())` returns a named vector, while `lmKfit()` returns a vector without names.

3. The computational complexity of `lmKfit()` is the sum of:

   - the complexity of the spectral decomposition, $O(n^3)$;
   - the complexity of computing $U^T y$, $O(n^2)$;
   - the complexity of computing $U^T X$, $O(n^2 p)$;
   - the complexity of fitting the weighted linear regression, $O(np^2)$.

   So the total complexity is $O(n^3 + n^2 p + np^2 + n^2)$, quadratic in the number of variables and cubic in the number of observations. We can check that is the case empirically by calling `lmKfit()` on random samples of varying size and comparing the resulting running times with a cubic fit.

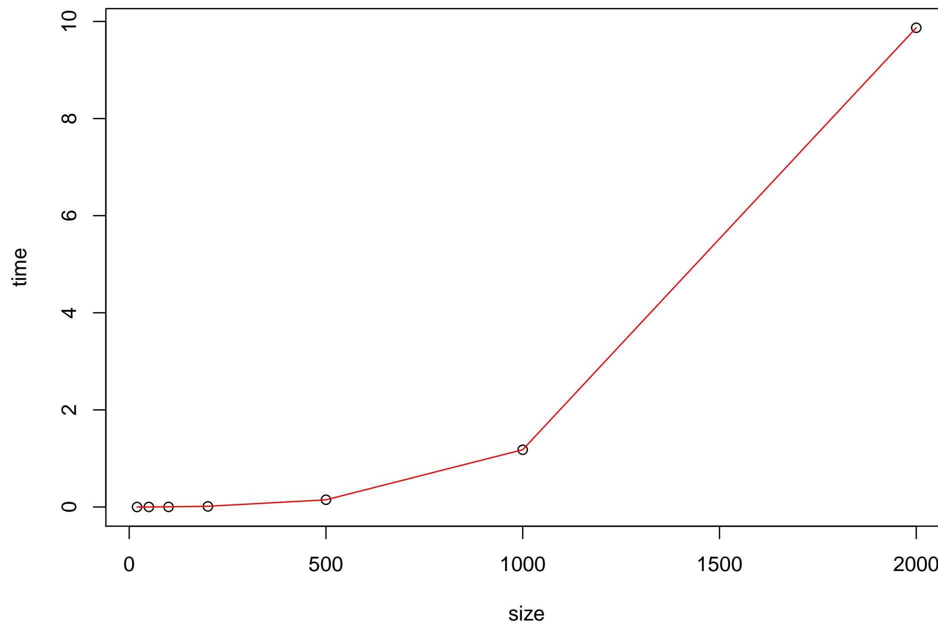
```r
n = 2000
h = 0.75
X = matrix(rnorm(5 * n), nrow = n, ncol = 5)
beta = (1:5)/5
K = diag(0.5, n) + 0.5
K[1, 2:n] = K[2:n, 1] = 0.75
epsilon = t(rmvnorm(1, sigma = h * K + (1 - h) * diag(n)))
y = as.vector(X %*% beta + epsilon)

size = c(20, 50, 100, 200, 500, 1000, 2000)
time = rep(0, length(size))
for (s in seq_along(size)) {

  ys = y[1:size[s]]
  Xs = X[1:size[s], ]
  Ks = K[1:size[s], 1:size[s]]

  time[s] = system.time(lmKfit(y = ys, X = Xs, h = h, K = Ks))["elapsed"]

}#FOR
plot(time ~ size)
cubic.fit = fitted(lm(time ~ size + I(size^2) + I(size^3)))
lines(y = cubic.fit, x = size, col = "red")
```



4. Note that in the implementation of `lmKhfit()` below, both X and y are premultiplied by U^T outside of `optimize()` to avoid redundant computations.

```r
lmKhfit = function(y, X, K) {

  # include the same arguments checks as in lmKfit() here.

  # compute the spectral decomposition of K.
  e = eigen(K)
  # rotate both X and y.
  Ut = t(e$vectors)
  yrot = Ut %*% y
  Xrot = Ut %*% X

  hfit = function(h, yrot, Xrot, K, return.beta) {

    # compute the weights from the eigenvalues and h.
    w = 1/(h * e$values + (1 - h))

    # estimate the betas usign the closed-form estimator for weighted
    # least squares.
    Xwt = t(Xrot * w)
    ywt = t(yrot * w)
    XwX = Xwt %*% Xrot
    Xwy = Xwt %*% yrot
    beta = solve(XwX) %*% Xwy

    # count how many times the weighted linear model is fitted, saving the
    # counter in the global environment.
    counter.lmKhfit <<- counter.lmKhfit + 1

    if (return.beta)
      return(list(h = h, beta = as.vector(beta)))
    else
      return(sum((y - X %*% beta)^2 / w))

  }#HFIT

  hopt = optimize(f = hfit, interval = c(0, 1),
           yrot = yrot, Xrot = Xrot, K = K, return.beta = FALSE)$minimum

  # refit the model with the optimal value of h and return.
  hfit(h = hopt, yrot = yrot, Xrot = Xrot, K = K, return.beta = TRUE)

}#LMKHFIT
```

5. The parallel implementation `parlmKhfit()` follows from `lmKhfit()`.

```r
parlmKhfit = function(y, X, K, cluster, stepping = 1/100) {

  # include the same arguments checks as in lmKfit() here.

  # compute the spectral decomposition of K.
  e = eigen(K)
  # rotate both X and y.
  Ut = t(e$vectors)
  yrot = Ut %*% y
  Xrot = Ut %*% X

  hfit = function(h, yrot, Xrot, K, return.beta = FALSE) {

    # compute the weights from the eigenvalues and h.
    w = 1/(h * e$values + (1 - h))

    # estimate the betas usign the closed-form estimator for weighted
    # least squares.
    Xwt = t(Xrot * w)
    ywt = t(yrot * w)
    XwX = Xwt %*% Xrot
    Xwy = Xwt %*% yrot
    beta = solve(XwX) %*% Xwy

    if (return.beta)
      return(list(h = h, beta = as.vector(beta)))
    else
      return(sum((y - X %*% beta)^2 / w))

  }#HFIT

  hgrid = seq(from = 0.01, to = 0.899, by = stepping)
  rss = parSapply(cluster, hgrid, hfit, yrot = yrot, Xrot = Xrot, K = K)
  hopt = hgrid[which.min(structure(rss, names = hgrid))]

  # count how many times the weighted linear model is fitted, saving the
  # counter in the global environment.
  counter.parlmKhfit <<- length(hgrid)

  # refit the model with the optimal value of h and return.
  hfit(h = hopt, yrot = yrot, Xrot = Xrot, K = K, return.beta = TRUE)

}#PARLMKHFIT
```

We can check that they give the same solution using some of the random data we have previously generated.

```r
library(parallel)
cl = makeCluster(2)

counter.lmKhfit = counter.parlmKhfit = 0

parlmKhfit(y = y[1:500], X = X[1:500, ], K = K[1:500, 1:500],
  cluster = cl, stepping = 1/200)

## $h
## [1] 0.815
##
## $beta
## [1] 0.155 0.425 0.583 0.836 0.978

lmKhfit(y = y[1:500], X = X[1:500, ], K = K[1:500, 1:500])

## $h
## [1] 0.814
##
## $beta
## [1] 0.155 0.425 0.583 0.836 0.978
```

6. `lmKhfit()` fits fewer weighted linear regressions than `parlmKhfit()`, because `optimize()` essentially searches by bisection:

```r
counter.lmKhfit

## [1] 14

counter.parlmKhfit

## [1] 178
```

So, for `parlmKhfit()` to be faster it must be run over more than $178/14 \approx 13$ cores. In that case each slave process fits fewer than 14 weighted regressions, and assuming overhead is negligible `parlmKhfit()` will be faster than `lmKhfit()`.