Statistical Programming, Week-Long Practical

Marco Scutari

April 24, 2017

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 : Model Selection for the Elastic Net

The *elastic net* is a penalised linear regression model that combines an L_1 penalty with an L_2 penalty to produce regularised estimates of the regression coefficients $\boldsymbol{\beta}$. In a frequentist setting, those estimates are computed by minimising the following penalised least squares problem:

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \left\{ (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + \lambda_1 \sum_{i=0}^{p} |\beta_i| + \lambda_2 \sum_{i=0}^{p} \beta_i^2 \right\}, \qquad \lambda_1, \lambda_2 \geqslant 0.$$

The λ_1 and λ_2 penalties are tuning parameters for which optimal values can be chosen by, say, 10-fold cross-validation. However, the tuning should be performed over *both* λ_1 and λ_2 *simultaneously* to pick the optimal model; their effects on the least squares minimisation interact and cannot be separated, so tuning λ_1 and then tuning λ_2 for $\hat{\lambda}_1$ or vice versa is likely to give suboptimal results.

First, you should analyse the computational (time) complexity of solving the penalised least squares problem for a given (λ_1, λ_2).

For this practical I would also like you to implement in R a simple *greedy grid search* (also known as *hill climbing*) that performs the tuning according to the following algorithm. I suggest you to check that your implementation works using the `nki70` data set included in **penalized**, with the molecular markers are explanatory variables and `time` as a response. *Hint: it should take at most 1 minute to run from any sensible starting point, say* $\lambda_1, \lambda_2 \in [5, 50]$.

Input: initial values $(\lambda_1^{(0)}, \lambda_2^{(0)})$, stepping (l_1, l_2), tolerance ε; set $\lambda_1^{(0)} > l_1$ and $\lambda_2^{(0)} > l_2$.
Ouptut: optimal values $(\hat{\lambda}_1, \hat{\lambda}_2)$.

1. Use 10 runs of 10-fold cross validation to evaluate $(\lambda_1^0, \lambda_2^{(0)})$ in terms of average predictive correlation (*i.e.* taking the average of the 10 predictive correlations computed from the 10 runs). These values will be the initial candidate solution.

2. For $i = 1, \ldots$ until convergence:

   (a) Consider the models $(\lambda_1^{(i-1)} + \{-l_1, 0, +l_1\}, \lambda_2^{(i-1)} + \{-l_2, 0, +l_2\})$, which are the "neighbours" of the current candidate solution on a grid with stepping (l_1, l_2).

   (b) Evaluate each of those models using 10 rounds of 10-fold cross validation and the average predictive correlation as in step 1.

i. If no model has predictive correlation that is greater than the predictive correlation for the current candidate $(\lambda_1^{(i-1)}, \lambda_2^{(i-1)})$ by at least ε, return the current candidate as $(\hat{\lambda}_1, \hat{\lambda}_2)$.

ii. Otherwise, select the model with the best predictive correlation as the new candidate $(\lambda_1^{(i)}, \lambda_2^{(i)})$.

(c) If $\lambda_1^{(i)} < l_1$, set $l_1 = \lambda_1^{(i)}/2$.

(d) If $\lambda_2^{(i)} < l_2$, set $l_2 = \lambda_2^{(i)}/2$.

You are allowed to use the **penalized** package, which implements the elastic net. A good implementation of the greedy grid search will also leverage the **parallel** package to speed up the tuning; you should discuss which parts can be executed in parallel, and which cannot, and provide an overview of various parallel implementation strategies beyond those you will implement in your code. You should not use any function that implements parts of the search, such as those in the **caret** package or any automated parameter tuning function that searches for the optimal $(\hat{\lambda}_1, \hat{\lambda}_2)$. (`cvl()` is obviously OK.)

Finally, I would like you to discuss and motivate any other choices you made to implement the greedy grid search in R in terms of data structures (*e.g.* lists vs arrays, etc.) and in terms of code organisation (*e.g.* factoring code into functions, etc.). *Hint: a naive implementation of the pseudocode above will compute many things over and over, and would result in much copy-and-pasted code, both of which should be avoided.*

Solution to Exercise

A sequential implementation of the greedy grid search is as follows. It is split in three functions:

- `generate.neighbours()`, which implements step 2(a);
- `eval.model.xval()` which implements steps 1 and 2(b);
- `hill.climbing()` which implements the whole algorithm.

The code implements a cache that stores the average predictive correlations, using the (λ_1, λ_2) as a key; parameter values that have not been evaluated yet are first introduced in the cache with a value of `NA` for the average predictive correlations. Then the cache is scanned and the `NA`s replaced with the estimated average predictive correlations.

```
library(penalized)

generate.neighbours = function(lambda, step) {

  # generate a 3x3 grid with right stepping
  nbrs = expand.grid(lambda1 = lambda[1] + c(-1, 0, 1) * step[1],
                     lambda2 = lambda[2] + c(-1, 0, 1) * step[2])
  # remove the central point (it's the current model, not a neighbour).
  nbrs = nbrs[-5, ]

  return(nbrs)

}#GENERATE.NEIGHBOURS

eval.model.xval = function(response, penalized, lambda) {

  cv.cor = numeric(10)

  for (i in seq_along(cv.cor)) {

    # perform cross-validation.
    pred = cvl(response, penalized = penalized, lambda1 = lambda[1],
            lambda2 = lambda[2], fold = 10, model = "linear", trace = FALSE)
    # compute the averaged cross-validated correlation.
    cv.cor[i] = cor(response, pred$predictions[, "mu"])

  }#FOR

  return(mean(cv.cor))

}#EVAL.MODEL.XVAL
```

```
hill.climbing = function(response, penalized, start, step, tol = 0.002) {

  # allocate the cache.
  score.cache =
    data.frame(lambda1 = numeric(0), lambda2 = numeric(0), COR = numeric(0))
  # fit the model for the initial lambdas.
  cur.cor = eval.model.xval(response = response, penalized = penalized,
              lambda = start)
  # save it in the cache.
  score.cache = rbind(score.cache, data.frame(lambda1 = start[1],
                  lambda2 = start[2], COR = cur.cor))

  repeat {

    cat("@ current lambdas: (", start[1], ",", start[2],
      ") with COR =", cur.cor, "\n")
    # halve stepping if the steps are too wide.
    if (any(start <= step))
      step[step >= start] = start[step >= start] / 2
    # generate the lambdas of the neighbouring models.
    nbrs = generate.neighbours(lambda = start, step = step)
    # look up in the score cache.
    nbrs = merge(nbrs, score.cache, all.x = TRUE)
    # fit the models not found in the score cache.
    for (i in seq(nrow(nbrs))) {

      # this model has already been explored, skip.
      if (!is.na(nbrs[i, "COR"]))
        next

      nbrs[i, "COR"] =
        eval.model.xval(response = response, penalized = penalized,
          lambda = as.numeric(nbrs[i, c("lambda1", "lambda2")]))

    }#FOR
    # merge back new scores into the cache.
    score.cache = merge(score.cache, nbrs, all = TRUE)
    # if the best lambdas are smaller than the candidate, return.
    best = nbrs[which.max(nbrs[, "COR"]), ]
    if (as.numeric(best[, "COR"]) < cur.cor + tol)
        break
    # update the current cross-validated correlation and candidate solution.
    start = as.numeric(best[, c("lambda1", "lambda2")])
    cur.cor = as.numeric(best[, "COR"])

  }#REPEAT

  return(list(lambda = start, cor = cur.cor, models = score.cache))

}#HILL.CLIMBING
```

The `hill.climbing()` function can be shown to work as follows.

```
data(nki70)
hill.climbing(response = nki70[, "time"], penalized = nki70[, 8:77],
  start = c(10, 10), step = c(5, 5), tol = 0.01)

## @ current lambdas: ( 10 , 10 ) with COR = 0.274
## @ current lambdas: ( 5 , 5 ) with COR = 0.321
## @ current lambdas: ( 5 , 2.5 ) with COR = 0.344
## $lambda
## [1] 5.0 2.5
##
## $cor
## [1] 0.344
##
## $models
##    lambda1 lambda2   COR
## 1      2.5    1.25 0.318
## 2      2.5    2.50 0.340
## 3      2.5    3.75 0.348
## 4      2.5    5.00 0.339
## 5      2.5    7.50 0.326
## 6      5.0    1.25 0.340
## 7      5.0    2.50 0.344
## 8      5.0    3.75 0.324
## 9      5.0    5.00 0.321
## 10     5.0    7.50 0.321
## 11     5.0   10.00 0.301
## 12     5.0   15.00 0.299
## 13     7.5    1.25 0.325
## 14     7.5    2.50 0.305
## 15     7.5    3.75 0.302
## 16     7.5    5.00 0.309
## 17     7.5    7.50 0.290
## 18    10.0    5.00 0.273
## 19    10.0   10.00 0.274
## 20    10.0   15.00 0.251
## 21    15.0    5.00 0.253
## 22    15.0   10.00 0.232
## 23    15.0   15.00 0.213
```

There are a number of ways to use parallel computing to speed up `hill.climbing()`. Clearly, steps much be executed sequentially since each step requires the candidate values from the previous step; but all the model evaluations within each step can be performed in parallel. The algorithm therefore displays coarse-grained parallelism.

One simple solution is to execute cross-validation runs in parallel in `eval.model.xval()` as follows. This is preferable to evaluating different (λ_1, λ_2) in parallel, since there are fewer than 10 in each step and therefore the maximum possible speed-up is smaller.

```
library(parallel)

eval.model.xval = function(response, penalized, lambda, slaves = 2) {

  FUN = function(id, response, penalized, lambda) {

    library(penalized)
    pred = cvl(response, penalized = penalized, lambda1 = lambda[1],
            lambda2 = lambda[2], fold = 10, model = "linear", trace = FALSE)
    cor(response, pred$predictions[, "mu"])

  }#FUN

  cl = makeCluster(slaves)
  cv.cor = parSapply(cl, 1:10, FUN, response = response,
             penalized = penalized, lambda = lambda)
  stopCluster(cl)

  return(mean(cv.cor))

}#EVAL.MODEL.XVAL
```

A much more scalable solution is to implement both runs and folds in parallel at the same time; this is possible because each fold in each run is independent of any other fold in any run. This makes it possible to use up to $10 \times 10 = 100$ slave processes as opposed to only 10.

```
eval.model.xval = function(response, penalized, lambda, slaves = 2) {

  cv.cor = numeric(10)

  FUN = function(fold, response, penalized, lambda) {

    library(penalized)

    penalized_train = penalized[-fold, ]
    response_train = response[-fold]
    penalized_test = penalized[fold, ]
    response_test = response[fold]

    model = penalized(response_train, penalized = penalized_train,
              lambda1 = lambda[1], lambda2 = lambda[2], trace = FALSE)

    pred = predict(model, penalized = penalized_test)

    return(data.frame(PRED = pred[, "mu"], OBS = response_test))

  }#FUN

  folds = replicate(10, split(sample(nrow(penalized)), seq_len(10)))

  cl = makeCluster(slaves)
  xval = parLapply(cl, folds, FUN, response = response,
             penalized = penalized, lambda = lambda)
  stopCluster(cl)

  for (i in 0:9)
    cv.cor[i] = cor(do.call("rbind", xval[10 * i + 1:10]))[2]

  return(mean(cv.cor))

}#EVAL.MODEL.XVAL
```

As for the time complexity of the penalized least squares optimisation problem,

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \left\{ (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + \lambda_1 \sum_{i=0}^{p} |\beta_i| + \lambda_2 \sum_{i=0}^{p} \beta_i^2 \right\}, \qquad \lambda_1, \lambda_2 \geqslant 0.$$

we have that

1. computing $\mathbf{X}\boldsymbol{\beta}$ is $O(np)$;
2. computing $Z = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}$ is $O(n)$;
3. computing $Z^T Z$ is $O(n)$;
4. computing $\sum_i |\beta_i|$ and $\sum_i \beta_i^2$ is $O(p)$.

from the analysis of the ordinary least squares complexity in the course material. So the time complexity of each evaluation is $O(np)$. The overall time complexity can be taken to be that of the LARS algorithm, $O(np^2)$, or $O(p^3)$ if we take $n \ll p$.