Robust Regression with Particle Swarm Optimisation and Differential Evolution

Enrico Schumann
es@enricoschumann.net

1 Introduction

We provide a code example for a robust regression problem; for more details, please see Gilli et al. [2011]. (The vignette builds on the script comparisonLMS.R)

2 Data and settings

We start by attaching the NMOF package and fixing a seed. We will use the function lqs from the MASS package [Venables and Ripley, 2002], so we attach that package as well.

```r
> library("NMOF")
> library("MASS")
> set.seed(11223344)
```

We will use an artificial data set with \( n \) observations and \( p \) regressors, created with the function `createData`.

```r
> createData <- function(n, p, constant = TRUE, 
|     sigma = 2, oFrac = 0.1) {
|     X <- array(rnorm(n * p), dim = c(n, p))
|     if (constant)
|         X[, 1L] <- 1L
|     b <- rnorm(p)
|     y <- X %*% b + rnorm(n)*0.5
|     n0 <- ceiling(oFrac*n)
|     when <- sample.int(n, n0)
|     X[when, -1L] <- X[when, -1L] + rnorm(n0, sd = sigma)
|     list(X = X, y = y, outliers = when)
| }
```

The function also takes arguments `constant` (logical: should the data-generating model contain a constant?); `sigma` (standard deviation of the outliers); and `oFrac` (fraction of outliers). The function evaluates to a list containing the regressors \( X \), the regressand \( y \) and a list of the outliers.

We put \( X \) and \( y \) into the list `Data`. We also add the scalar \( h \), which gives the order statistic of the squared residuals to be minimised. Note that we put `as.vector(y)` into `Data` so that the vector gets ‘recycled’ in the objective function.

```r
> n <- 100L  ## number of observations
> p <- 10L   ## number of regressors
> constant <- TRUE; sigma <- 5; oFrac <- 0.1
> h <- 75L   ## ... or use something like floor((n+1)/2)
> aux <- createData(n, p, constant, sigma, oFrac)
> X <- aux$X; y <- aux$y
> Data <- list(y = as.vector(y), X = X, h = h)
```

The outliers, added in blue, are often visible.

```r
> par(bty = "n", las = 1, tck = 0.01, mar = c(4,4,1,1))
> plot(X[, 2L], type = "h", ylab = "X values", xlab = "observation")
> lines(aux$outliers, X[aux$outliers, 2L], type = "p", pch = 21, col = "blue", bg = "blue")
```
Two example objective functions, Least Trimmed Squares (LTS) and Least Quantile of Squares (LQS).
Note that they are identical except for their last line.

```r
> OF <- function(param, Data) {
  X <- Data$X; y <- Data$y
  aux <- y - X %*% param
  aux <- aux * aux
  aux <- apply(aux, 2L, sort, partial = Data$h)
  colSums(aux[1:Data$h, ]) ## LTS
}

> OF <- function(param, Data) {
  X <- Data$X; y <- Data$y
  aux <- y - X %*% param
  aux <- aux * aux
  aux <- apply(aux, 2L, sort, partial = Data$h)
  aux[Data$h, ] ## LQS
}
```

Both functions are vectorised. They work with a single solution (param would be a vector) or a whole population (param would be a matrix; each column would be one solution).

### 3 Using DE and PSO

We run DE and PSO. We compare the result with \texttt{lqs}.

```r
> popsize <- 100L; generations <- 500L
> ps <- list(min = rep(-10,p),
  max = rep( 10,p),
  c1 = 0.9,
  c2 = 0.9,
  iner = 0.9,
  initV = 1,
  nP = popsize,
  nQ = generations,
  maxV = 5,
  loopOF = FALSE,
  printBar = FALSE,
  printDetail = FALSE)

> de <- list(min = rep(-10,p),
  max = rep( 10,p),
  nP = popsize,
  nQ = generations,
  maxV = 5,
  loopOF = FALSE,
  printBar = FALSE,
  printDetail = FALSE)
```
nG = generations,
F = 0.7,
CR = 0.9,
loopOF = FALSE,
printBar = FALSE,
printDetail = FALSE)

> system.time(solPS <- PSopt(OF = OF, algo = ps, Data = Data))

    user  system elapsed
  1.002   0.000   1.002

> system.time(solDE <- DEopt(OF = OF, algo = de, Data = Data))

    user  system elapsed
  0.993   0.000   0.993

> if (require("MASS", quietly = TRUE)) {
    system.time(test1 <- lqs(y ~ X[,,-1L], adjust = TRUE,
                             nsamp = 100000L, method = "lqs",
                             quantile = h))
    res1 <- sort((y - X %*% as.matrix(coef(test1)))^2)[h]
} else
    res1 <- NA

> res2 <- sort((y - X %*% as.matrix(solPS$xbest))^2)[h]
> res3 <- sort((y - X %*% as.matrix(solDE$xbest))^2)[h]

> cat("lqs: ", res1, "\n",
    "PSopt: ", res2, "\n",
    "DEopt: ", res3, "\n", sep = "")

lqs: 0.38106
PSopt: 0.29676
DEopt: 0.25894

To demonstrate the advantage of a vectorised objective function, we can compare it with looping over
the solutions. We first set loopOF to TRUE, so we actually loop over the solutions. (We also reduce the
number of objective function evaluations since we do not care about the actual solution, only about speed
of computation.)

> popsize <- 100L; generations <- 20L
> de$nP <- popsize; de$nG <- generations
> ps$nP <- popsize; ps$nG <- generations
> de$loopOF <- TRUE; ps$loopOF <- TRUE
> t1ps <- system.time(solPS <- PSopt(OF = OF, algo = ps, Data = Data))
> t1de <- system.time(solDE <- DEopt(OF = OF, algo = de, Data = Data))

To evaluate the objective function in one step, we loopOF to FALSE.

> de$loopOF <- FALSE; ps$loopOF <- FALSE
> t2ps <- system.time(solPS <- PSopt(OF = OF, algo = ps, Data = Data))
> t2de <- system.time(solDE <- DEopt(OF = OF, algo = de, Data = Data))

Speedup:

> t1ps[[3L]]/t2ps[[3L]]  ## PS

[1] 2.675

> t1de[[3L]]/t2de[[3L]]  ## DE

[1] 2.65
References
