Robust Regression with Particle Swarm Optimisation and Differential Evolution

Enrico Schumann

October 20, 2011

Abstract

A brief tutorial on using Differential Evolution and Particle Swarm Optimisation to estimate a regression model.

1 Introduction

We provide a code example for a robust regression problem. The purpose of this vignette is to provide the code in a convenient way; 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 package.

```
> require("NMOF")
> require("MASS")
> set.seed(11223344)
```

We will use the function `lqs` from the `MASS` package [Venables and Ripley, 2002]. We will use an artificial data set with `n` observations and `p` regressors, created with the function `createData`.

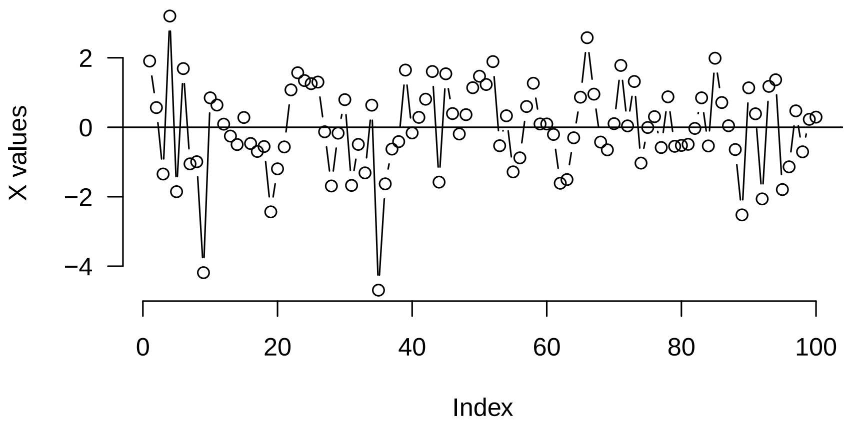
```
> createData <- function(n, p, constant = TRUE, sigma = 2, oFrac = 0.1) {
+     X <- array(rnorm(n * p), dim = c(n, p))
+     if (constant)
+         X[, 1L] <- 1L
+     b <- rnorm(p)
+     y <- X %*% b + rnorm(n) * 0.5
+     nO <- ceiling(oFrac * n)
+     when <- sample.int(n, nO)
+     X[when,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

We start by creating some artifial data. We collect `X` and `y` in 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.

```
> n <- 100L
> p <- 10L
> constant <- TRUE
> sigma <- 3
> oFrac <- 0.1
> h <- 75L
> 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 are visible.

```
> par(bty = "n", las = 1)
> plot(X[, 2L], type = "b", ylab = "X values")
> abline(h = 0)
```



Two example objective functions, Least Trimmed Squares (LTS) and Least Quantile of Squares (LQS). Note that they are almost identical.

```
> 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, ])
+ }
> 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, ]
+ }
```

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 `lqs`.

```
> popsize <- 100L
> generations <- 500L
> ps <- list(min = rep(-10, p), max = rep(10, p), c1 = 0.5, c2 = 1.1,
+     iner = 0.9, initV = 1, nP = popsize, nG = generations, maxV = 5,
+     loopOF = FALSE, printBar = FALSE, printDetail = FALSE)
> de <- list(min = rep(-10, p), max = rep(10, p), nP = popsize,
+     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 
   1.89    0.00    1.89 
```

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

```
   user  system elapsed 
   1.90    0.00    1.91 
```

```
> 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])
```

```
[1] 0.2633539
```

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

```
[1] 0.2798945
```

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

```
lqs:   0.3807335
PSopt: 0.2633539
DEopt: 0.2798945
```

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.)

```
> popsize <- 20L
> generations <- 150L
> 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)))
```

```
   user  system elapsed 
   0.41    0.00    0.41 
```

```
> (t1de <- system.time(solDE <- DEopt(OF = OF, algo = de, data = data)))
```

```
   user  system elapsed 
   0.39    0.00    0.39 
```

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)))
```

```
   user  system elapsed 
   0.12    0.00    0.12 
```

```
> (t2de <- system.time(solDE <- DEopt(OF = OF, algo = de, data = data)))
```

```
   user  system elapsed 
   0.14    0.00    0.14 
```

Speedup:

```
> t1ps[[3L]]/t2ps[[3L]]
```

```
[1] 3.416667
```

```
> t1de[[3L]]/t2de[[3L]]
```

```
[1] 2.785714
```

References

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier, 2011.

William N. Venables and Brian D. Ripley. *Modern Applied Statistics with S*. Springer, 4th edition, 2002. URL http://www.stats.ox.ac.uk/pub/MASS4.