Package 'psqn'

November 27, 2020

Type Package

Title Partially Separable Quasi-Newton

Version 0.1.4

Maintainer Benjamin Christoffersen <boennecd@gmail.com>

Description Provides quasi-Newton methods to minimize partially separable functions. The methods are largely described by Nocedal and Wright (2006) <doi:10.1007/978-0-387-40065-5>.

License GPL-2

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

Depends R (>= 3.5.0)

URL https://github.com/boennecd/psqn

BugReports https://github.com/boennecd/psqn/issues

LinkingTo Rcpp, testthat

Imports Rcpp

Suggests R.rsp, rmarkdown, RcppArmadillo, bench, testthat, numDeriv, lbfgsb3c, lbfgs, Matrix

VignetteBuilder R.rsp

SystemRequirements C++11

NeedsCompilation yes

Author Benjamin Christoffersen [cre, aut]

Repository CRAN

Date/Publication 2020-11-27 02:20:07 UTC

R topics documented:

psqn *Partially Separable Function Optimization*

Description

Optimization method for specially structured partially separable functions.

Usage

```
psqn(
  par,
  fn,
  n_ele_func,
  rel_eps = 1e-08,
  max_it = 100L,
  n_threads = 1L,
  c1 = 1e-04,
  c2 = 0.9,
  use_bfgs = TRUE,
  trace = 0L,
  cg_tol = 0.5,
  strong_wolfe = TRUE,
  env = NULL,
  max_cg = 0L,
  pre_method = 1L
)
```

Arguments

par	Initial values for the parameters. It is a concatenated vector of the global parameters and all the private parameters.
fn	Function to compute the element functions and their derivatives. Each call computes on element function. See the examples section.
n_ele_func	Number of element functions.
rel_eps	Relative convergence threshold.
max_it	Maximum number of iterations.
n_threads	Number of threads to use.
c1, c2	Thresholds for the Wolfe condition.
use_bfgs	Logical for whether to use BFGS updates or SR1 updates.
trace	Integer where larger values gives more information during the optimization.
cg_tol	Threshold for the conjugate gradient method.
strong_wolfe	TRUE if the strong Wolfe condition should be used.
env	Environment to evaluate fn in. NULL yields the global environment.

max_cg	maximum number of conjugate gradient iterations in each iteration. Use zero if there should not be a limit.
pre_method	preconditioning method in conjugate gradient method. zero yields no preconditioning and one yields diagonal preconditioning.

Details

The function follows the method described by Nocedal and Wright (2006) and mainly what is described in Section 7.4. Details are provided in the psqn vignette. See `vignette("psqn",package = "psqn")`.

The partially separable function we consider are special in that the function to be minimized is a sum of so-called element functions which only depend on few shared (global) parameters and some private parameters which are particular to each element function.

The optimization function is also available in C++ as a header-only library. Using C++ may reduce the computation time substantially. See the vignette in the package for examples.

Value

An object with the following elements:

par	the estimated global and private parameters.
value	function value at `par`.
info	information code. 0 implies convergence. -1 implies that the maximum number iterations is reached. -2 implies that the conjugate gradient method failed. -3 implies that the line search failed. -4 implies that the user interrupted the optimization.
counts	An integer vector with the number of function evaluations, gradient evaluations, and the number of conjugate gradient iterations.
convergence	`TRUE` if `info == 0`.

References

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization* (2nd ed.). Springer.

Examples

```
# example with inner problem in a Taylor approximation for a mixed GLMM as
# in the vignette

# assign model parameters, number of random effects, and fixed effects
q <- 2 # number of private parameters per cluster
p <- 1 # number of global parameters
beta <- sqrt((1:p) / sum(1:p))
Sigma <- diag(q)

# simulate a data set
set.seed(66608927)
n_clusters <- 20L # number of clusters
sim_dat <- replicate(n_clusters, {
```

```
  n_members <- sample.int(8L, 1L) + 2L
  X <- matrix(runif(p * n_members, -sqrt(6 / 2), sqrt(6 / 2)),
              p)
  u <- drop(rnorm(q) %*% chol(Sigma))
  Z <- matrix(runif(q * n_members, -sqrt(6 / 2 / q), sqrt(6 / 2 / q)),
              q)
  eta <- drop(beta %*% X + u %*% Z)
  y <- as.numeric((1 + exp(-eta))^(-1) > runif(n_members))

  list(X = X, Z = Z, y = y, u = u, Sigma_inv = solve(Sigma))
}, simplify = FALSE)

# evalutes the negative log integrand.
#
# Args:
#   i cluster/element function index.
#   par the global and private parameter for this cluster. It has length
#       zero if the number of parameters is requested. That is, a 2D integer
#       vector the number of global parameters as the first element and the
#       number of private parameters as the second element.
#   comp_grad logical for whether to compute the gradient.
r_func <- function(i, par, comp_grad){
  dat <- sim_dat[[i]]
  X <- dat$X
  Z <- dat$Z

  if(length(par) < 1)
    # requested the dimension of the parameter
    return(c(global_dim = NROW(dat$X), private_dim = NROW(dat$Z)))

  y <- dat$y
  Sigma_inv <- dat$Sigma_inv

  beta <- par[1:p]
  uhat <- par[1:q + p]
  eta <- drop(beta %*% X + uhat %*% Z)
  exp_eta <- exp(eta)

  out <- -sum(y * eta) + sum(log(1 + exp_eta)) +
    sum(uhat * (Sigma_inv %*% uhat)) / 2
  if(comp_grad){
    d_eta <- -y + exp_eta / (1 + exp_eta)
    grad <- c(X %*% d_eta,
              Z %*% d_eta + dat$Sigma_inv %*% uhat)
    attr(out, "grad") <- grad
  }

  out
}

# optimize the log integrand
res <- psqn(par = rep(0, p + q * n_clusters), fn = r_func,
            n_ele_func = n_clusters)
```

```
head(res$par, p)              # the estimated global parameters
tail(res$par, n_clusters * q) # the estimated private parameters

# compare with
beta
c(sapply(sim_dat, "[[", "u"))
```

---

psqn_bfgs *BFGS Implementation Used Internally in the psqn Package*

Description

The method seems to mainly differ from `optim` by the line search method. This version uses the interpolation method with a zoom phase using cubic interpolation as described by Nocedal and Wright (2006).

Usage

```
psqn_bfgs(
  par,
  fn,
  gr,
  rel_eps = 1e-08,
  max_it = 100L,
  c1 = 1e-04,
  c2 = 0.9,
  trace = 0L,
  env = NULL
)
```

Arguments

`par`	Initial values for the parameters.
`fn`	Function to evaluate the function to be minimized.
`gr`	Gradient of `fn`. Should return the function value as an attribute called `"value"`.
`rel_eps`	Relative convergence threshold.
`max_it`	Maximum number of iterations.
`c1`	Thresholds for the Wolfe condition.
`c2`	Thresholds for the Wolfe condition.
`trace`	Integer where larger values gives more information during the optimization.
`env`	Environment to evaluate `fn` and `gr` in. `NULL` yields the global environment.

Value

An object like the object returned by `psqn`.

References

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization* (2nd ed.). Springer.

Examples

```
# declare function and gradient from the example from help(optim)
fn <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
gr <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
     200 *      (x2 - x1 * x1))
}

# we need a different function for the method in this package
gr_psqn <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  out <- c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
           200 *      (x2 - x1 * x1))
  attr(out, "value") <- 100 * (x2 - x1 * x1)^2 + (1 - x1)^2
  out
}

# we get the same
optim    (c(-1.2, 1), fn, gr, method = "BFGS")
psqn_bfgs(c(-1.2, 1), fn, gr_psqn)

# compare the computation time
system.time(replicate(1000,
                      optim    (c(-1.2, 1), fn, gr, method = "BFGS")))
system.time(replicate(1000,
                      psqn_bfgs(c(-1.2, 1), fn, gr_psqn)))
```

Index