

Package ‘quadprogXT’

October 13, 2022

Title Quadratic Programming with Absolute Value Constraints

Version 0.0.5

Description Extends the quadprog package to solve quadratic programs with absolute value constraints and absolute values in the objective function.

Imports quadprog

License GPL (>= 2)

Encoding UTF-8

LazyData true

Suggests tinytest

RoxygenNote 6.1.1

NeedsCompilation no

Author Bob Harlow [aut, cre]

Maintainer Bob Harlow <rharlow86@gmail.com>

Repository CRAN

Date/Publication 2020-01-15 12:30:02 UTC

R topics documented:

convertToCompact	2
normalizeConstraints	2
solveQPXT	3
Index	6

convertToCompact	<i>'Sparsify' constraint matrix</i>
------------------	-------------------------------------

Description

'Sparsify' constraint matrix

Usage

convertToCompact(Amat)

Arguments

Amat	a constraint matrix as defined in solve.QP
------	--

Value

a list with two elements: Amat and Aind as necessary to be passed to solve.QP.compact

See Also

quadprog::solve.QP
quadprog::solve.QP.compact

normalizeConstraints	<i>Normalize constraint matrix</i>
----------------------	------------------------------------

Description

it is not uncommon for quadprog to fail when there are large differences in 2-norm between the columns of the constraint matrix (Amat). It is possible to alleviate this issue in some cases by normalizing the constraints (and their boundaries, defined by bvec).

Usage

normalizeConstraints(Amat, bvec)

Arguments

Amat	constraint matrix as defined by solve.QP
bvec	constraints as defined by solve.QP

Value

a list with two elements: Amat and bvec that contain the normalized constraints.

See Also

quadprog::solve.QP
 quadprog::solve.QP.compact

solveQPXT	<i>Solve a quadratic program with absolute values in constraints & objective</i>
-----------	--

Description

solveQPXT allows for absolute value constraints and absolute values in the objective. buildQP builds a parameter list that can then be passed to quadprog::solve.QP.compact or quadprog::solve.QP directly if desired by the user. solveQPXT by default implicitly takes advantage of sparsity in the constraint matrix and can improve numerical stability by normalizing the constraint matrix. For the rest of the documentation, assume that Dmat is $n \times n$.

The solver solves the following problem (each * corresponds to matrix multiplication):

```

min:
-t(dvec) * b + 1/2 t(b) * Dmat * b +
-t(dvecPosNeg) * c(b_positive, b_negative) +
-t(dvecPosNegDelta) * c(deltab_positive, deltab_negative)

s.t.
t(Amat) * b >= bvec
t(AmatPosNeg) * c(b_positive, b_negative) >= bvecPosNeg
t(AmatPosNegDelta) * c(deltab_positive, deltab_negative) >= bvecPosNegDelta
b_positive, b_negative >= 0,
b = b_positive - b_negative
deltab_positive, deltab_negative >= 0,
b - b0 = deltab_positive - deltab_negative

```

Usage

```

solveQPXT(...)

buildQP(Dmat, dvec, Amat, bvec, meq = 0, factorized = FALSE,
  AmatPosNeg = NULL, bvecPosNeg = NULL, dvecPosNeg = NULL,
  b0 = NULL, AmatPosNegDelta = NULL, bvecPosNegDelta = NULL,
  dvecPosNegDelta = NULL, tol = 1e-08, compact = TRUE,
  normalize = TRUE)

```

Arguments

...	parameters to pass to buildQP when calling solveQPXT
Dmat	matrix appearing in the quadratic function to be minimized.
dvec	vector appearing in the quadratic function to be minimized.
Amat	matrix defining the constraints under which we want to minimize the quadratic function.
bvec	vector holding the values of b_0 (defaults to zero).
meq	the first meq constraints are treated as equality constraints, all further as inequality constraints (defaults to 0).
factorized	logical flag: if TRUE, then we are passing R^{-1} (where $D = R^T R$) instead of the matrix D in the argument Dmat.
AmatPosNeg	2n x k matrix of constraints on the positive and negative part of b
bvecPosNeg	k length vector of thresholds to the constraints in AmatPosNeg
dvecPosNeg	k * 2n length vector of loadings on the positive and negative part of b, respectively
b0	a starting point that describes the 'current' state of the problem such that constraints and penalty on absolute changes in the decision variable from a starting point can be incorporated. b0 is an n length vector. Note that b0 is NOT a starting point for the optimization - that is handled implicitly by quadprog.
AmatPosNegDelta	2n x l matrix of constraints on the positive and negative part of a change in b from a starting point, b0.
bvecPosNegDelta	l length vector of thresholds to the constraints in AmatPosNegDelta
dvecPosNegDelta	l * 2n length vector of loadings in the objective function on the positive and negative part of changes in b from a starting point of b0.
tol	tolerance along the diagonal of the expanded Dmat for slack variables
compact	logical: if TRUE, it is assumed that we want to use solve.QP.compact to solve the problem, which handles sparsity.
normalize	logical: should constraint matrix be normalized

Details

In order to handle constraints on b_{positive} and b_{negative} , slack variables are introduced. The total number of parameters in the problem increases by the following amounts:

If all the new parameters (those not already used by quadprog) remain NULL, the problem size does not increase and `quadprog::solve.QP(.compact)` is called after normalizing the constraint matrix and converting to a sparse matrix representation by default.

If `AmatPosNeg`, `bvecPosNeg` or `dvecPosNeg` are not null, the problem size increases by n. If `AmatPosNegDelta` or `dvecPosNegDelta` are not null, the problem size increases by n. This results in a potential problem size of up to $3 * n$. Despite the potential large increases in problem size, the underlying solver is written in Fortran and converges quickly for problems involving even hundreds

of parameters. Additionally, it has been the author's experience that solutions solved via the convex quadprog are much more stable than those solved by other methods (e.g. a non-linear solver).

Note that due to the fact that the constraints are by default normalized, the original constraint values the user passed will may not be returned by buildQP.

Examples

```
##quadprog example"
Dmat      <- matrix(0,3,3)
diag(Dmat) <- 1
dvec      <- c(0,5,0)
Amat      <- matrix(c(-4,-3,0,2,1,0,0,-2,1),3,3)
bvec      <- c(-8,2,0)
qp <- quadprog::solve.QP(Dmat,dvec,Amat,bvec=bvec)
qpXT <- solveQPXT(Dmat,dvec,Amat,bvec=bvec)
range(qp$solution - qpXT$solution)

N <- 10
set.seed(2)
cr <- matrix(runif(N * N, 0, .05), N, N)
diag(cr) <- 1
cr <- (cr + t(cr)) / 2
set.seed(3)
sigs <- runif(N, min = .02, max = .25)
set.seed(5)
dvec <- runif(N, -.1, .1)
Dmat <- sigs %o% sigs * cr
Amat <- cbind(diag(N), diag(N) * -1)
bvec <- c(rep(-1, N), rep(-1, N))
resBase <- solveQPXT(Dmat, dvec, Amat, bvec)
##absolute value constraint on decision variable:
res <- solveQPXT(Dmat, dvec, Amat, bvec,
AmatPosNeg = matrix(rep(-1, 2 * N)), bvecPosNeg = -1)
sum(abs(res$solution[1:N]))

## penalty of L1 norm
resL1Penalty <- solveQPXT(Dmat, dvec, Amat, bvec, dvecPosNeg = -.005 * rep(1, 2 * N))
sum(abs(resL1Penalty$solution[1:N]))

## constraint on amount decision variable can vary from a starting point
b0 <- rep(.15, N)
thresh <- .25
res <- solveQPXT(Dmat, dvec, Amat, bvec, b0 = b0,
AmatPosNegDelta = matrix(rep(-1, 2 * N)), bvecPosNegDelta = -thresh)
sum(abs(res$solution[1:N] - b0))

##use buildQP, then call solve.QP.compact directly
qp <- buildQP(Dmat, dvec, Amat, bvec, b0 = b0,
AmatPosNegDelta = matrix(rep(-1, 2 * N)), bvecPosNegDelta = -thresh)
res2 <- do.call(quadprog::solve.QP.compact, qp)
range(res$solution - res2$solution)
```

Index

`buildQP (solveQPXT)`, 3
`convertToCompact`, 2
`normalizeConstraints`, 2
`solveQPXT`, 3