Package: mixsqp (via r-universe)

October 28, 2026

Encoding UTF-8

Type Package

Version 0.3-54

Date 2023-12-20

Title Sequential Quadratic Programming for Fast Maximum-Likelihood Estimation of Mixture Proportions

URL https://github.com/stephenslab/mixsqp

BugReports https://github.com/stephenslab/mixsqp/issues

Depends R (>= 3.3.0)

Description Provides an optimization method based on sequential quadratic programming (SQP) for maximum likelihood estimation of the mixture proportions in a finite mixture model where the component densities are known. The algorithm is expected to obtain solutions that are at least as accurate as the state-of-the-art MOSEK interior-point solver (called by function ``KWDual'' in the 'REBayes' package), and they are expected to arrive at solutions more quickly when the number of samples is large and the number of mixture components is not too large. This implements the ``mix-SQP'' algorithm, with some improvements, described in Y. Kim, P. Carbonetto, M. Stephens & M. Anitescu (2020) <DOI:10.1080/10618600.2019.1689985>.

License MIT + file LICENSE

Imports utils, stats, irlba, Rcpp (>= 0.12.15)

Suggests testthat, knitr, rmarkdown

LinkingTo Rcpp, RcppArmadillo

LazyData true

NeedsCompilation yes

VignetteBuilder knitr

RoxygenNote 7.1.2

Author Youngseok Kim [aut], Peter Carbonetto [aut, cre], Mihai Anitescu [aut], Matthew Stephens [aut], Jason Willwerscheid [ctb], Jean Morrison [ctb]

Maintainer Peter Carbonetto <peter.carbonetto@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2023-12-21 02:43:08 UTC

RemoteUrl https://github.com/cran/mixsqp

RemoteRef HEAD

RemoteSha c8e49ae42dcd96d898c586da6b024298c92692ee

Contents

mixsqp-package 2
mixobjective 2
mixsqp 3
simulatemixdata 7
tacks 9

Index **10**

`mixsqp-package` *mixsqp: Sequential Quadratic Programming for Fast Maximum-Likelihood Estimation of Mixture Proportions*

Description

Provides optimization algorithms based on sequential quadratic programming (SQP) for maximum likelihood estimation of the mixture proportions in a finite mixture model where the component densities are known. To learn more, visit `https://github.com/stephenslab/mixsqp`, and see the help for function `mixsqp`.

`mixobjective` *Compute objective optimized by mixsqp.*

Description

See `mixsqp` for a full description of the objective function optimized by the mix-SQP algorithm.

Usage

```
mixobjective(L, x, w = rep(1, nrow(L)))
```

Arguments

L	Matrix specifying the optimization problem to be solved. In the context of mixture-model fitting, L[j,k] should be the value of the kth mixture component density at the jth data point. L should be a numeric matrix with at least two columns, with all entries being non-negative and finite (and not missing). Further, no column should be entirely zeros. For large matrices, it is preferrable that the matrix is stored in double-precision; see storage.mode.
x	The point at which the objective is evaluated in mixobjective; see argument x0 in mixsqp for details.
w	An optional numeric vector, with one entry for each row of L, specifying the "weights" associated with the rows of L. All weights must be finite, non-negative and not missing. Internally, the weights are normalized to sum to 1, which does not change the problem, but does change the value of the objective function reported. By default, all weights are equal.

Value

The value of the objective at x. If any entry of L %*% x is less than or equal to zero, Inf is returned.

See Also

mixsqp

---

mixsqp *Maximum-likelihood estimation of mixture proportions using SQP*

Description

The mixsqp function uses a Sequential Quadratic Programming (SQP) algorithm to find the maximum likelihood estimates of mixture proportions in a (finite) mixture model. More generally, mixsqp solves the corresponding constrained, convex optimization problem, which is given below (see 'Details'). See 'References' for more details about the SQP algorithm.

Usage

```
mixsqp(
  L,
  w = rep(1, nrow(L)),
  x0 = rep(1, ncol(L)),
  log = FALSE,
  control = list()
)

mixsqp_control_default()
```

Arguments

L	Matrix specifying the optimization problem to be solved. In the context of mixture-model fitting, L[j,k] should be the value of the kth mixture component density at the jth data point. L should be a numeric matrix with at least two columns, with all entries being non-negative and finite (and not missing). In some cases, it is easier or more natural to compute log(L); for example, it is often easier to compute the log-likelihood rather than the likelihood. Setting log = TRUE will tell mixsqp to interpret this input as the logarithm of the data matrix. Note that, for large matrices, it is preferrable that the matrix is stored in double-precision; see storage.mode.
w	An optional numeric vector, with one entry for each row of L, specifying the "weights" associated with the rows of L. All weights must be finite, non-negative and not missing. Internally, the weights are normalized to sum to 1, which does not change the problem, but does change the value of the objective function reported. By default, all weights are equal.
x0	An optional numeric vector providing an initial estimate of the solution to the optimization problem. It should contain only finite, non-missing, non-negative values, and all entries of L %*% x0 must be greater than zero (to ensure that the objective evaluates to a finite value at x0). The vector will be normalized to sum to 1. By default, x0 is the vector with all equal values.
log	When log = TRUE, the input matrix L is interpreted as containing the logarithm of the data matrix.
control	A list of parameters controlling the behaviour of the optimization algorithm. See 'Details'.

Details

mixsqp solves the following optimization problem. Let L be a matrix with n rows and m columns containing only non-negative entries, and let $w = (w_1, \ldots, w_n)$ be a vector of non-negative "weights". mixsqp computes the value of vector $x = (x_1, \ldots, x_m)$ minimizing the following objective function,

$$f(x) = -\sum_{j=1}^{n} w_j \log(\sum_{k=1}^{m} L_{jk} x_k),$$

subject to the constraint that x lie within the simplex; that is, the entries of x are non-negative and sum to 1. Implicitly, there is an additional constraint $L * x > 0$ in order to ensure that the objective has a finite value. In practice, this constraint only needs to be checked for the initial estimate to ensure that it holds for all subsequent iterates.

If all weights are equal, solving this optimization problem corresponds to finding the maximum-likelihood estimate of the mixture proportions x given n independent data points drawn from a mixture model with m components. In this case, L_{jk} is the likelihood for mixture component k and data point j.

The Expectation Maximization (EM) algorithm can be used to solve this optimization problem, but it is intolerably slow in many interesting cases, and mixsqp is much faster.

A special feature of this optimization problem is that the gradient of the objective does not change with re-scaling; for example, if all the entries of matrix L are multiplied by 100, the gradient does

not change. A practical benefit of this property is that the optimization algorithm will behave similarly irrespective of the scale of L; for example, the same value for the convergence tolerance convtol.sqp will have the same effect at different scales.

A related feature is that the solution to the optimization problem is invariant to rescaling the rows of L; for example, the solution will remain the same after all the entries in a row of L are multiplied by 10. A simple normalization scheme divides each row by the largest entry in the row so that all entries of L are at most 1: L <- L / apply(L,1,max) Occasionally, it can be helpful to normalize the rows when some of the entries are unusually large or unusually small. This can help to avoid numerical overflow or underflow errors.

The SQP algorithm is implemented using the Armadillo C++ linear algebra library, which can automatically take advantage of multithreaded matrix computations to speed up mixsqp for large L matrices, but only when R has been configured with a multithreaded BLAS/LAPACK library (e.g., OpenBLAS).

A "debugging mode" is provided to aid in reproducing convergence failures or other issues. When activated, mixsqp will generate an .RData file containing the exact mixsqp inputs, and will stop execution upon convergence failure. To activate the debugging mode, run options(mixsqp.debug.mode = TRUE) prior to calling mixsqp. By default, the output file is mixsqp.RData; the file can be changed by setting the "mixsqp.debug.file" global option.

The control argument is a list in which any of the following named components will override the default optimization algorithm settings (as they are defined by mixsqp_control_default):

normalize.rows When normalize.rows = TRUE, the rows of the data matrix L are automatically scaled so that the largest entry in each row is 1. This is the recommended setting for better stability of the optimization. When log = TRUE, this setting is ignored becase the rows are already normalized. Note that the objective is computed on the original (unnormalized) matrix to make the results easier to interpret.

tol.svd Setting used to determine rank of truncated SVD approximation for L. The rank of the truncated singular value decomposition is determined by the number of singular values surpassing tol.svd. When tol.svd = 0 or when L has 4 or fewer columns, all computations are performed using full L matrix.

convtol.sqp A small, non-negative number specifying the convergence tolerance for SQP algorithm; convergence is reached when the maximum dual residual in the Karush-Kuhn-Tucker (KKT) optimality conditions is less than or equal to convtol.sqp. Smaller values will result in more stringent convergence criteria and more accurate solutions, at the expense of greater computation time. Note that changes to this tolerance parameter may require respective changes to convtol.activeset and/or zero.threshold.searchdir to obtain reliable convergence.

convtol.activeset A small, non-negative number specifying the convergence tolerance for the active-set step. Smaller values will result in higher quality search directions for the SQP algorithm but possibly a greater per-iteration computational cost. Note that changes to this tolerance parameter can affect how reliably the SQP convergence criterion is satisfied, as determined by convtol.sqp.

zero.threshold.solution A small, non-negative number used to determine the "active set"; that is, it determines which entries of the solution are exactly zero. Any entries that are less than or equal to zero.threshold.solution are considered to be exactly zero. Larger values of zero.threshold.solution may lead to speedups for matrices with many columns, at the (slight) risk of prematurely zeroing some co-ordinates.

`zero.threshold.searchdir` A small, non-negative number used to determine when the search direction in the active-set step is considered "small enough". Note that changes to this tolerance parameter can affect how reliably the SQP convergence criterion is satisfied, as determined by `convtol.sqp`, so choose this parameter carefully.

`suffdecr.linesearch` This parameter determines how stringent the "sufficient decrease" condition is for accepting a step size in the backtracking line search. Larger values will make the condition more stringent. This should be a positive number less than 1.

`stepsizereduce` The multiplicative factor for decreasing the step size in the backtracking line search. Smaller values will yield a faster backtracking line search at the expense of a less fine-grained search. Should be a positive number less than 1.

`minstepsize` The smallest step size accepted by the line search step. Should be a number greater than 0 and at most 1.

`identity.contrib.increase` When the Hessian is not positive definite, a multiple of the identity is added to the Hessian to ensure a unique search direction. The factor for increasing the identity contribution in this modified Hessian is determined by this control parameter.

`eps` A small, non-negative number that is added to the terms inside the logarithms to sidestep computing logarithms of zero. This prevents numerical problems at the cost of introducing a small inaccuracy in the solution. Increasing this number may lead to faster convergence but possibly a less accurate solution.

`maxiter.sqp` Maximum number of SQP iterations to run before reporting a convergence failure; that is, the maximum number of quadratic subproblems that will be solved by the active-set method.

`maxiter.activeset` Maximum number of active-set iterations taken to solve each of the quadratic subproblems. If `NULL`, the maximum number of active-set iterations is set to `min(20,1 + ncol(L))`.

`numiter.em` Number of expectation maximization (EM) updates to perform prior to running mix-SQP. Although EM can often be slow to converge, this "pre-fitting" step can help to obtain a good initial estimate for mix-SQP at a small cost.

`verbose` If `verbose = TRUE`, the algorithm's progress and a summary of the optimization settings are printed to the console. The algorithm's progress is displayed in a table with one row per SQP (outer loop) iteration, and with the following columns: "iter", the (outer loop) SQP iteration; "objective", the value of the objective function (see $f(x)$) at the current estimate of the solution, x; "max(rdual)", the maximum "dual residual" in the Karush-Kuhn-Tucker (KKT) conditions, which is used to monitor convergence (see `convtol.sqp`); "nnz", the number of non-zero co-ordinates in the current estimate, as determined by `zero.threshold.solution`; "max.diff", the maximum difference in the estimates between two successive iterations; "nqp", the number of (inner loop) active-set iterations taken to solve the quadratic subproblem; "nls", the number of iterations in the backtracking line search.

Value

A list object with the following elements:

`x` If the SQP algorithm converges, this is the solution to the convex optimization problem. If the algorithm fails to converge, it is the best estimate of the solution achieved by the algorithm. Note that if the SQP algorithm terminates before

reaching the solution, x may not satisfy the equality constraint; that is, the entries of x may not sum to 1.

`value` The value of the objective function, $f(x)$, at x.

`grad` The gradient of the objective function at x.

`hessian` The Hessian of the objective function at x. The truncated SVD approximation of L is used to compute the Hessian when it is also used for mix-SQP.

`status` A character string describing the status of the algorithm upon termination.

`progress` A data frame containing more detailed information about the algorithm's progress. The data frame has one row per SQP iteration. For an explanation of the columns, see the description of the `verbose` control parameter in 'Details'. Missing values (`NA`'s) in the last row indicate that these quantities were not computed because convergence was reached before computing them. Also note that the storage of these quantities in the `progress` data frame is slightly different than in the console output (when `verbose = TRUE`) as the console output shows some quantities that were computed after the convergence check in the previous iteration.

References

Y. Kim, P. Carbonetto, M. Stephens and M. Anitescu (2020). A fast algorithm for maximum likelihood estimation of mixture proportions using sequential quadratic programming. *Journal of Computational and Graphical Statistics* **29**, 261-273. doi: 10.1080/10618600.2019.1689985

See Also

`mixobjective`

Examples

```
set.seed(1)
n <- 1e5
m <- 10
w <- rep(1,n)/n
L <- simulatemixdata(n,m)$L
out.mixsqp <- mixsqp(L,w)
f <- mixobjective(L,out.mixsqp$x,w)
print(f,digits = 16)
```

---

simulatemixdata *Create likelihood matrix from simulated data set*

Description

Simulate a data set, then compute the conditional likelihood matrix under a univariate normal likelihood and a mixture-of-normals prior. This models a simple nonparametric Empirical Bayes method applied to simulated data.

Usage

```
simulatemixdata(
  n,
  m,
  simtype = c("n", "nt"),
  log = FALSE,
  normalize.rows = !log
)
```

Arguments

n	Positive integer specifying the number of samples to generate and, consequently, the number of rows of the likelihood matrix L.
m	Integer 2 or greater specifying the number of mixture components.
simtype	The type of data to simulate. If simtype = "n", simulate n random numbers from a mixture of three univariate normals with mean zero and standard deviation 1, 3 and 6. If simtype = "nt", simulate from a mixture of three univariate normals (with zero mean and standard deviations 1, 3 and 5), and a t-distribution with 2 degrees of freedom.
log	If log = TRUE, return the
normalize.rows	If normalize.rows = TRUE, normalize the rows of the likelihood matrix so that the largest entry in each row is 1. The maximum-likelihood estimate of the mixture weights should be invariant to this normalization, and can improve the numerical stability of the optimization.

Value

simulatemixdata returns a list with three list elements:

x	The vector of simulated random numbers (it has length n).
s	The standard deviations of the mixture components in the mixture-of-normals prior. The rules for selecting the standard deviations are based on the autoselect.mixsd function from the ashr package.
L	The n x m conditional likelihood matrix, in which individual entries (i,j) of the likelihood matrix are given by the normal density function with mean zero and variance 1 + s[j]^2. If normalize.rows = TRUE, the entries in each row are normalized such that the larger entry in each row is 1. If log = TRUE, the matrix of log-likelihoods is returned.

Examples

```
# Generate the likelihood matrix for a data set with 1,000 samples
# and a nonparametric Empirical Bayes model with 20 mixture
# components.
dat <- simulatemixdata(1000,20)
```

tacks *Beckett & Diaconis tack rolling example.*

Description

This data set contains the likelihood matrix and weights for the Beckett-Diaconis tacks example, in which the data are modeled using a binomial mixture. These data were generated by running the "Bmix1" demo from the REBayes package, and saving the arguments passed to `KWDual`, as well as the (normalized) solution returned by the `KWDual` call.

Format

`tacks` is a list with the following elements:

L 9 x 299 likelihood matrix.

w Numeric vector of length 9 specifying the weights associated with the rows of `L`.

x Solution provided by the `KWDual` solver.

Examples

```
# The optimal solution for the tack example is extremely sparse.
data(tacks)
plot(tacks$x,type = "l",col = "royalblue")
```

Index

∗ **data**
    tacks, 9

mixobjective, 2, *7*
mixsqp, *2*, 3, *3*
mixsqp-package, 2
mixsqp_control_default *(*mixsqp*)*, 3

simulatemixdata, 7
storage.mode, *3*, *4*

tacks, 9