Package ‘rnnmf’

July 23, 2025

Maintainer Steven E. Pav <shabbychef@gmail.com>

Version 0.3.0

Date 2024-10-30

License LGPL-3

Title Regularized Non-Negative Matrix Factorization

BugReports https://github.com/shabbychef/rnnmf/issues

Description
A proof of concept implementation of regularized non-negative matrix factorization optimization. A non-negative matrix factorization factors non-negative matrix Y approximately as L R, for non-negative matrices L and R of reduced rank. This package supports such factorizations with weighted objective and regularization penalties. Allowable regularization penalties include L1 and L2 penalties on L and R, as well as non-orthogonality penalties. This package provides multiplicative update algorithms, which are a modification of the algorithm of Lee and Seung (2001) <http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf>, as well as an additive update derived from that multiplicative update. See also Pav (2004) <doi:10.48550/arXiv.2410.22698>.

Depends R (>= 3.0.2)

Imports Matrix

Suggests testthat, dplyr, ggplot2, scales, viridis, knitr

URL https://github.com/shabbychef/rnnmf

VignetteBuilder knitr

Collate 'aurnmf.r' 'gaurnmf.r' 'giqpm.r' 'murnmf.r' 'rnnmf-package.r'

RoxygenNote 7.3.2

NeedsCompilation no

Author Steven E. Pav [aut, cre] (ORCID: <https://orcid.org/0000-0002-4197-6195>)

Repository CRAN

Date/Publication 2024-11-04 10:40:02 UTC

Contents

`aurnmf` *nmf*.

Description

Additive update Non-negative matrix factorization with regularization.

Usage

```
aurnmf(
  Y,
  L,
  R,
  W_0R = NULL,
  W_0C = NULL,
  lambda_1L = 0,
  lambda_1R = 0,
  lambda_2L = 0,
  lambda_2R = 0,
  gamma_2L = 0,
  gamma_2R = 0,
  tau = 0.1,
  annealing_rate = 0.01,
  check_optimal_step = TRUE,
  zero_tolerance = 1e-12,
  max_iterations = 1000L,
  min_xstep = 1e-09,
  on_iteration_end = NULL,
  verbosity = 0
)
```

Arguments

- `Y` — an $r \times c$ matrix to be decomposed. Should have non-negative elements; an error is thrown otherwise.
- `L` — an $r \times d$ matrix of the initial estimate of L. Should have non-negative elements; an error is thrown otherwise.
- `R` — an $d \times c$ matrix of the initial estimate of R. Should have non-negative elements; an error is thrown otherwise.
- `W_0R` — the row space weighting matrix. This should be a positive definite non-negative symmetric $r \times r$ matrix. If omitted, it defaults to the properly sized identity matrix.
- `W_0C` — the column space weighting matrix. This should be a positive definite non-negative symmetric $c \times c$ matrix. If omitted, it defaults to the properly sized identity matrix.
- `lambda_1L` — the scalar ℓ_1 penalty for the matrix L. Defaults to zero.
- `lambda_1R` — the scalar ℓ_1 penalty for the matrix R. Defaults to zero.
- `lambda_2L` — the scalar ℓ_2 penalty for the matrix L. Defaults to zero.
- `lambda_2R` — the scalar ℓ_2 penalty for the matrix R. Defaults to zero.
- `gamma_2L` — the scalar ℓ_2 penalty for non-orthogonality of the matrix L. Defaults to zero.
- `gamma_2R` — the scalar ℓ_2 penalty for non-orthogonality of the matrix R. Defaults to zero.
- `tau` — the starting shrinkage factor applied to the step length. Should be a value in $(0, 1)$.
- `annealing_rate` — the rate at which we scale the shrinkage factor towards 1. Should be a value in $[0, 1)$.
- `check_optimal_step` — if TRUE, we attempt to take the optimal step length in the given direction. If not, we merely take the longest feasible step in the step direction.
- `zero_tolerance` — values of x less than this will be 'snapped' to zero. This happens at the end of the iteration and does not affect the measurement of convergence.
- `max_iterations` — the maximum number of iterations to perform.
- `min_xstep` — the minimum L-infinity norm of the step taken. Once the step falls under this value, we terminate.
- `on_iteration_end` — an optional function that is called at the end of each iteration. The function is called as `on_iteration_end(iteration=iteration, Y=Y, L=L, R=R, Lstep=Lstep, Rstep=Rstep, ...)`
- `verbosity` — controls whether we print information to the console.

Details

Attempts to factor given non-negative matrix Y as the product LR of two non-negative matrices. The objective function is Frobenius norm with ℓ_1 and ℓ_2 regularization terms. We seek to minimize the objective

$$\frac{1}{2}tr((Y-LR)'W_{0R}(Y-LR)W_{0C})+\lambda_{1L}|L|+\lambda_{1R}|R|+\frac{\lambda_{2L}}{2}tr(L'L)+\frac{\lambda_{2R}}{2}tr(R'R)+\frac{\gamma_{2L}}{2}tr((L'L)(11'-I))+\frac{\gamma_{2R}}{2}tr((R'R$$

subject to $L \geq 0$ and $R \geq 0$ elementwise, where $|A|$ is the sum of the elements of A and $tr(A)$ is the trace of A.

The code starts from initial estimates and iteratively improves them, maintaining non-negativity. This implementation uses the Lee and Seung step direction, with a correction to avoid divide-by-zero. The iterative step is optionally re-scaled to take the steepest descent in the step direction.

Value

a list with the elements

L The final estimate of L.

R The final estimate of R.

Lstep The infinity norm of the final step in L.

Rstep The infinity norm of the final step in R.

iterations The number of iterations taken.

converged Whether convergence was detected.

Note

This package provides proof of concept code which is unlikely to be fast or robust, and may not solve the optimization problem at hand. User assumes all risk.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References

Merritt, Michael, and Zhang, Yin. "Interior-point Gradient Method for Large-Scale Totally Nonnegative Least Squares Problems." Journal of Optimization Theory and Applications 126, no 1 (2005): 191–202. https://scholarship.rice.edu/bitstream/handle/1911/102020/TR04-08.pdf

Pav, S. E. "An Iterative Algorithm for Regularized Non-negative Matrix Factorizations." Forthcoming. (2024)

Lee, Daniel D. and Seung, H. Sebastian. "Algorithms for Non-negative Matrix Factorization." Advances in Neural Information Processing Systems 13 (2001): 556–562. http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

See Also

gaurnmf, murnmf.

Examples

```
nr <- 100
nc <- 20
dm <- 4

randmat <- function(nr,nc,...) { matrix(pmax(0,runif(nr*nc,...)),nrow=nr) }
```

```
 set.seed(1234)
 real_L <- randmat(nr,dm)
 real_R <- randmat(dm,nc)
 Y <- real_L %*% real_R
# without regularization
 objective <- function(Y, L, R) { sum((Y - L %*% R)^2) }
 objective(Y,real_L,real_R)

 L_0 <- randmat(nr,dm)
 R_0 <- randmat(dm,nc)
 objective(Y,L_0,R_0)
 out1 <- aurnmf(Y, L_0, R_0, max_iterations=5e3L,check_optimal_step=FALSE)
 objective(Y,out1$L,out1$R)
# with L1 regularization on one side
 out2 <- aurnmf(Y, L_0, R_0, lambda_1L=0.1, max_iterations=5e3L,check_optimal_step=FALSE)
# objective does not suffer because all mass is shifted to R
 objective(Y,out2$L,out2$R)
list(L1=sum(out1$L),R1=sum(out1$R),L2=sum(out2$L),R2=sum(out2$R))
sum(out2$L)
# with L1 regularization on both sides
 out3 <- aurnmf(Y, L_0, R_0, lambda_1L=0.1,lambda_1R=0.1,
     max_iterations=5e3L,check_optimal_step=FALSE)
# with L1 regularization on both sides, raw objective suffers
 objective(Y,out3$L,out3$R)
list(L1=sum(out1$L),R1=sum(out1$R),L3=sum(out3$L),R3=sum(out3$R))


# example showing how to use the on_iteration_end callback to save iterates.
max_iterations <- 5e3L
it_history <<- rep(NA_real_, max_iterations)
quadratic_objective <- function(Y, L, R) { sum((Y - L %*% R)^2) }
on_iteration_end <- function(iteration, Y, L, R, ...) {
  it_history[iteration] <<- quadratic_objective(Y,L,R)
}
out1b <- aurnmf(Y, L_0, R_0, max_iterations=max_iterations, on_iteration_end=on_iteration_end)


# should work on sparse matrices too.
if (require(Matrix)) {
 real_L <- randmat(nr,dm,min=-1)
 real_R <- randmat(dm,nc,min=-1)
 Y <- as(real_L %*% real_R, "sparseMatrix")
 L_0 <- as(randmat(nr,dm,min=-0.5), "sparseMatrix")
 R_0 <- as(randmat(dm,nc,min=-0.5), "sparseMatrix")
 out1 <- aurnmf(Y, L_0, R_0, max_iterations=1e2L,check_optimal_step=TRUE)
}
```

---

gaurnmf *gaurnmf* .

Description

Additive update Non-negative matrix factorization with regularization, general form.

Usage

```
gaurnmf(
  Y,
  L,
  R,
  W_0R = NULL,
  W_0C = NULL,
  W_1L = 0,
  W_1R = 0,
  W_2RL = 0,
  W_2CL = 0,
  W_2RR = 0,
  W_2CR = 0,
  tau = 0.1,
  annealing_rate = 0.01,
  check_optimal_step = TRUE,
  zero_tolerance = 1e-12,
  max_iterations = 1000L,
  min_xstep = 1e-09,
  on_iteration_end = NULL,
  verbosity = 0
)
```

Arguments

Y	an $r \times c$ matrix to be decomposed. Should have non-negative elements; an error is thrown otherwise.
L	an $r \times d$ matrix of the initial estimate of L. Should have non-negative elements; an error is thrown otherwise.
R	an $d \times c$ matrix of the initial estimate of R. Should have non-negative elements; an error is thrown otherwise.
W_0R	the row space weighting matrix. This should be a positive definite non-negative symmetric $r \times r$ matrix. If omitted, it defaults to the properly sized identity matrix.
W_0C	the column space weighting matrix. This should be a positive definite non-negative symmetric $c \times c$ matrix. If omitted, it defaults to the properly sized identity matrix.
W_1L	the ℓ_1 penalty matrix for the matrix R. If a scalar, corresponds to that scalar times the all-ones matrix. Defaults to all-zeroes matrix, which is no penalty term.
W_1R	the ℓ_1 penalty matrix for the matrix L. If a scalar, corresponds to that scalar times the all-ones matrix. Defaults to all-zeroes matrix, which is no penalty term.

`W_2RL` the ℓ_2 row penalty matrix for the matrix L. If a scalar, corresponds to that scalar times the identity matrix. Can also be a list, in which case `W_2CL` must be a list of the same length. The list should consist of ℓ_2 row penalty matrices. Defaults to all-zeroes matrix, which is no penalty term.

`W_2CL` the ℓ_2 column penalty matrix for the matrix L. If a scalar, corresponds to that scalar times the identity matrix. Can also be a list, in which case `W_2RL` must be a list of the same length. The list should consist of ℓ_2 column penalty matrices. Defaults to all-zeroes matrix, which is no penalty term.

`W_2RR` the ℓ_2 row penalty matrix for the matrix R. If a scalar, corresponds to that scalar times the identity matrix. Can also be a list, in which case `W_2CR` must be a list of the same length. The list should consist of ℓ_2 row penalty matrices. Defaults to all-zeroes matrix, which is no penalty term.

`W_2CR` the ℓ_2 column penalty matrix for the matrix R. If a scalar, corresponds to that scalar times the identity matrix. Can also be a list, in which case `W_2RR` must be a list of the same length. The list should consist of ℓ_2 column penalty matrices. Defaults to all-zeroes matrix, which is no penalty term.

`tau` the starting shrinkage factor applied to the step length. Should be a value in $(0, 1)$.

`annealing_rate` the rate at which we scale the shrinkage factor towards 1. Should be a value in $[0, 1)$.

`check_optimal_step`
if TRUE, we attempt to take the optimal step length in the given direction. If not, we merely take the longest feasible step in the step direction.

`zero_tolerance` values of x less than this will be 'snapped' to zero. This happens at the end of the iteration and does not affect the measurement of convergence.

`max_iterations` the maximum number of iterations to perform.

`min_xstep` the minimum L-infinity norm of the step taken. Once the step falls under this value, we terminate.

`on_iteration_end`
an optional function that is called at the end of each iteration. The function is called as `on_iteration_end(iteration=iteration, Y=Y, L=L, R=R, Lstep=Lstep, Rstep=Rstep, ...)`

`verbosity` controls whether we print information to the console.

Details

Attempts to factor given non-negative matrix Y as the product LR of two non-negative matrices. The objective function is Frobenius norm with ℓ_1 and ℓ_2 regularization terms. We seek to minimize the objective

$$\frac{1}{2}tr((Y-LR)'W_{0R}(Y-LR)W_{0C})+tr(W_{1L}'L)+tr(W_{1R}'R)+\frac{1}{2}\sum_j tr(L'W_{2RLj}LW_{2CLj})+tr(R'W_{2RRj}RW_{2CRj}),$$

subject to $L \geq 0$ and $R \geq 0$ elementwise, where $tr(A)$ is the trace of A.

The code starts from initial estimates and iteratively improves them, maintaining non-negativity. This implementation uses the Lee and Seung step direction, with a correction to avoid divide-by-zero. The iterative step is optionally re-scaled to take the steepest descent in the step direction.

Value

a list with the elements

L The final estimate of L.

R The final estimate of R.

Lstep The infinity norm of the final step in L.

Rstep The infinity norm of the final step in R.

iterations The number of iterations taken.

converged Whether convergence was detected.

Note

This package provides proof of concept code which is unlikely to be fast or robust, and may not solve the optimization problem at hand. User assumes all risk.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References

Merritt, Michael, and Zhang, Yin. "Interior-point Gradient Method for Large-Scale Totally Nonnegative Least Squares Problems." Journal of Optimization Theory and Applications 126, no 1 (2005): 191–202. https://scholarship.rice.edu/bitstream/handle/1911/102020/TR04-08.pdf

Pav, S. E. "An Iterative Algorithm for Regularized Non-negative Matrix Factorizations." Forthcoming. (2024)

Lee, Daniel D. and Seung, H. Sebastian. "Algorithms for Non-negative Matrix Factorization." Advances in Neural Information Processing Systems 13 (2001): 556–562. http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

See Also

aurnmf

Examples

```
nr <- 20
nc <- 5
dm <- 2

randmat <- function(nr,nc,...) { matrix(pmax(0,runif(nr*nc,...)),nrow=nr) }
set.seed(1234)
real_L <- randmat(nr,dm+2)
real_R <- randmat(ncol(real_L),nc)
Y <- real_L %*% real_R
gram_it <- function(G) { t(G) %*% G }
W_0R <- gram_it(randmat(nr+5,nr))
W_0C <- gram_it(randmat(nc+5,nc))
```

```
wt_objective <- function(Y, L, R, W_0R, W_0C) {
  err <- Y - L %*% R
  0.5 * sum((err %*% W_0C) * (t(W_0R) %*% err))
}
matrix_trace <- function(G) {
  sum(diag(G))
}
wt_objective(Y,real_L,real_R,W_0R,W_0C)

L_0 <- randmat(nr,dm)
R_0 <- randmat(dm,nc)
wt_objective(Y,L_0,R_0,W_0R,W_0C)
out1 <- gaurnmf(Y, L_0, R_0, W_0R=W_0R, W_0C=W_0C,
        max_iterations=1e4L,check_optimal_step=FALSE)
wt_objective(Y,out1$L,out1$R,W_0R,W_0C)

W_1L <- randmat(nr,dm)
out2 <- gaurnmf(Y, out1$L, out1$R, W_0R=W_0R, W_0C=W_0C, W_1L=W_1L,
        max_iterations=1e4L,check_optimal_step=FALSE)
wt_objective(Y,out2$L,out2$R,W_0R,W_0C)

W_1R <- randmat(dm,nc)
out3 <- gaurnmf(Y, out2$L, out2$R, W_0R=W_0R, W_0C=W_0C, W_1R=W_1R,
        max_iterations=1e4L,check_optimal_step=FALSE)
wt_objective(Y,out3$L,out3$R,W_0R,W_0C)


# example showing how to use the on_iteration_end callback to save iterates.
max_iterations <- 1e3L
it_history <<- rep(NA_real_, max_iterations)
on_iteration_end <- function(iteration, Y, L, R, ...) {
  it_history[iteration] <<- wt_objective(Y,L,R,W_0R,W_0C)
}
out1b <- gaurnmf(Y, L_0, R_0, W_0R=W_0R, W_0C=W_0C,
 max_iterations=max_iterations, on_iteration_end=on_iteration_end, check_optimal_step=FALSE)


# should work on sparse matrices too.
if (require(Matrix)) {
 real_L <- randmat(nr,dm,min=-1)
 real_R <- randmat(dm,nc,min=-1)
 Y <- as(real_L %*% real_R, "sparseMatrix")
 L_0 <- as(randmat(nr,dm,min=-0.5), "sparseMatrix")
 R_0 <- as(randmat(dm,nc,min=-0.5), "sparseMatrix")
 out1 <- gaurnmf(Y, L_0, R_0, max_iterations=1e2L,check_optimal_step=TRUE)
}
```

---

giqpm *giqpm .*

Description

Generalized Iterative Quadratic Programming Method for non-negative quadratic optimization.

Usage

```
giqpm(
  Gmat,
  dvec,
  x0 = NULL,
  tau = 0.5,
  annealing_rate = 0.25,
  check_optimal_step = TRUE,
  mult_func = NULL,
  grad_func = NULL,
  step_func = NULL,
  zero_tolerance = 1e-09,
  max_iterations = 1000L,
  min_xstep = 1e-09,
  verbosity = 0
)
```

Arguments

`Gmat`	a representation of the matrix G.
`dvec`	a representation of the vector d.
`x0`	the initial iterate. If none given, we spawn one of the same size as `dvec`.
`tau`	the starting shrinkage factor applied to the step length. Should be a value in $(0, 1)$.
`annealing_rate`	the rate at which we scale the shrinkage factor towards 1. Should be a value in $[0, 1)$.
`check_optimal_step`	if TRUE, we attempt to take the optimal step length in the given direction. If not, we merely take the longest feasible step in the step direction.
`mult_func`	a function which takes matrix and vector and performs matrix multiplication. The default does this on matrix and vector input, but the user can implement this for some implicit versions of the problem.
`grad_func`	a function which takes matrix G, vector d, the current iterate x and the product Gx and is supposed to compute $Gx + d$. The default does this on matrix and vector input, but the user can implement this for some implicit versions of the problem.
`step_func`	a function which takes the vector gradient, the product Gx, the matrix G, vector d, vector x and the `mult_func` and produces a step vector. By default this step vector is the Lee-Seung step vector, namely $-(Gx + d) * x/d$, with Hadamard product and division.
`zero_tolerance`	values of x less than this will be ‘snapped’ to zero. This happens at the end of the iteration and does not affect the measurement of convergence.

`max_iterations`	the maximum number of iterations to perform.
`min_xstep`	the minimum L-infinity norm of the step taken. Once the step falls under this value, we terminate.
`verbosity`	controls whether we print information to the console.

Details

Iteratively solves the problem

$$\min_x \frac{1}{2} x^\top G x + d^\top x$$

subject to the elementwise constraint $x \geq 0$.

This implementation allows the user to specify methods to perform matrix by vector multiplication, computation of the gradient (which should be $Gx + d$), and computation of the step direction. By default we compute the optimal step in the given step direction.

Value

a list with the elements

x The final iterate.

iterations The number of iterations taken.

converged Whether convergence was detected.

Note

This package provides proof of concept code which is unlikely to be fast or robust, and may not solve the optimization problem at hand. User assumes all risk.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References

Pav, S. E. "An Iterative Algorithm for Regularized Non-negative Matrix Factorizations." Forthcoming. (2024)

Merritt, Michael, and Zhang, Yin. "Interior-point Gradient Method for Large-Scale Totally Nonnegative Least Squares Problems." Journal of Optimization Theory and Applications 126, no 1 (2005): 191–202. https://scholarship.rice.edu/bitstream/handle/1911/102020/TR04-08.pdf

Examples

```
set.seed(1234)
ssiz <- 100
preG <- matrix(runif(ssiz*(ssiz+20)),nrow=ssiz)
G <- preG %*% t(preG)
d <- - runif(ssiz)
y1 <- giqpm(G, d)
objective <- function(G, d, x) { as.numeric(0.5 * t(x) %*% (G %*% x) + t(x) %*% d) }
```

```
# this does not converge to an actual solution!
steepest_step_func <- function(gradf, ...) { return(-gradf) }
y2 <- giqpm(G, d, step_func = steepest_step_func)

scaled_step_func <- function(gradf, Gx, Gmat, dvec, x0, ...) { return(-gradf * abs(x0)) }
y3 <- giqpm(G, d, step_func = scaled_step_func)

sqrt_step_func <- function(gradf, Gx, Gmat, dvec, x0, ...) { return(-gradf * abs(sqrt(x0))) }
y4 <- giqpm(G, d, step_func = sqrt_step_func)

complementarity_stepfunc <- function(gradf, Gx, Gmat, dvec, x0, ...) { return(-gradf * x0) }
y5 <- giqpm(G, d, step_func = complementarity_stepfunc)

objective(G, d, y1$x)
objective(G, d, y2$x)
objective(G, d, y3$x)
objective(G, d, y4$x)
objective(G, d, y5$x)
```

---

murnmf *murnmf*.

Description

Multiplicative update Non-negative matrix factorization with regularization.

Usage

```
murnmf(
  Y,
  L,
  R,
  W_0R = NULL,
  W_0C = NULL,
  lambda_1L = 0,
  lambda_1R = 0,
  lambda_2L = 0,
  lambda_2R = 0,
  gamma_2L = 0,
  gamma_2R = 0,
  epsilon = 1e-07,
  max_iterations = 1000L,
  min_xstep = 1e-09,
  on_iteration_end = NULL,
  verbosity = 0
)
```

Arguments

`Y`	an $r \times c$ matrix to be decomposed. Should have non-negative elements; an error is thrown otherwise.
`L`	an $r \times d$ matrix of the initial estimate of L. Should have non-negative elements; an error is thrown otherwise.
`R`	an $d \times c$ matrix of the initial estimate of R. Should have non-negative elements; an error is thrown otherwise.
`W_0R`	the row space weighting matrix. This should be a positive definite non-negative symmetric $r \times r$ matrix. If omitted, it defaults to the properly sized identity matrix.
`W_0C`	the column space weighting matrix. This should be a positive definite non-negative symmetric $c \times c$ matrix. If omitted, it defaults to the properly sized identity matrix.
`lambda_1L`	the scalar ℓ_1 penalty for the matrix L. Defaults to zero.
`lambda_1R`	the scalar ℓ_1 penalty for the matrix R. Defaults to zero.
`lambda_2L`	the scalar ℓ_2 penalty for the matrix L. Defaults to zero.
`lambda_2R`	the scalar ℓ_2 penalty for the matrix R. Defaults to zero.
`gamma_2L`	the scalar ℓ_2 penalty for non-orthogonality of the matrix L. Defaults to zero.
`gamma_2R`	the scalar ℓ_2 penalty for non-orthogonality of the matrix R. Defaults to zero.
`epsilon`	the numerator clipping value.
`max_iterations`	the maximum number of iterations to perform.
`min_xstep`	the minimum L-infinity norm of the step taken. Once the step falls under this value, we terminate.
`on_iteration_end`	an optional function that is called at the end of each iteration. The function is called as `on_iteration_end(iteration=iteration, Y=Y, L=L, R=R, Lstep=Lstep, Rstep=Rstep, ...)`
`verbosity`	controls whether we print information to the console.

Details

This function uses multiplicative updates only, and may not optimize the nominal objective. It is also unlikely to achieve optimality. This code is for reference purposes and is not suited for usage other than research and experimentation.

Value

a list with the elements

L The final estimate of L.

R The final estimate of R.

Lstep The infinity norm of the final step in L.

Rstep The infinity norm of the final step in R.

iterations The number of iterations taken.

converged Whether convergence was detected.

Note

This package provides proof of concept code which is unlikely to be fast or robust, and may not solve the optimization problem at hand. User assumes all risk.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References

Merritt, Michael, and Zhang, Yin. "Interior-point Gradient Method for Large-Scale Totally Nonnegative Least Squares Problems." Journal of Optimization Theory and Applications 126, no 1 (2005): 191–202. https://scholarship.rice.edu/bitstream/handle/1911/102020/TR04-08.pdf

Pav, S. E. "An Iterative Algorithm for Regularized Non-negative Matrix Factorizations." Forthcoming. (2024)

Lee, Daniel D. and Seung, H. Sebastian. "Algorithms for Non-negative Matrix Factorization." Advances in Neural Information Processing Systems 13 (2001): 556–562. http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

See Also

aurnmf, gaurnmf

Examples

```
 nr <- 100
 nc <- 20
 dm <- 4

 randmat <- function(nr,nc,...) { matrix(pmax(0,runif(nr*nc,...)),nrow=nr) }
 set.seed(1234)
 real_L <- randmat(nr,dm)
 real_R <- randmat(dm,nc)
 Y <- real_L %*% real_R
# without regularization
 objective <- function(Y, L, R) { sum((Y - L %*% R)^2) }
 objective(Y,real_L,real_R)

 L_0 <- randmat(nr,dm)
 R_0 <- randmat(dm,nc)
 objective(Y,L_0,R_0)
 out1 <- murnmf(Y, L_0, R_0, max_iterations=5e3L)
 objective(Y,out1$L,out1$R)
# with L1 regularization on one side
 out2 <- murnmf(Y, L_0, R_0, max_iterations=5e3L,lambda_1L=0.1)
# objective does not suffer because all mass is shifted to R
 objective(Y,out2$L,out2$R)
list(L1=sum(out1$L),R1=sum(out1$R),L2=sum(out2$L),R2=sum(out2$R))
sum(out2$L)
# with L1 regularization on both sides
```

```
 out3 <- murnmf(Y, L_0, R_0, max_iterations=5e3L,lambda_1L=0.1,lambda_1R=0.1)
# with L1 regularization on both sides, raw objective suffers
 objective(Y,out3$L,out3$R)
list(L1=sum(out1$L),R1=sum(out1$R),L3=sum(out3$L),R3=sum(out3$R))


# example showing how to use the on_iteration_end callback to save iterates.
max_iterations <- 1e3L
it_history <<- rep(NA_real_, max_iterations)
quadratic_objective <- function(Y, L, R) { sum((Y - L %*% R)^2) }
on_iteration_end <- function(iteration, Y, L, R, ...) {
  it_history[iteration] <<- quadratic_objective(Y,L,R)
}
out1b <- murnmf(Y, L_0, R_0, max_iterations=max_iterations, on_iteration_end=on_iteration_end)


# should work on sparse matrices too, but beware zeros in the initial estimates
if (require(Matrix)) {
 real_L <- randmat(nr,dm,min=-1)
 real_R <- randmat(dm,nc,min=-1)
 Y <- as(real_L %*% real_R, "sparseMatrix")
 L_0 <- randmat(nr,dm)
 R_0 <- randmat(dm,nc)
 out1 <- murnmf(Y, L_0, R_0, max_iterations=1e2L)
}
```

`rnnmf-NEWS` *News for package 'rnnmf':*

Description

News for package 'rnnmf'

rnnmf Initial Version 0.3.0 (2024-10-30)

- first CRAN release.
- changed name from rnmf to rnnmf.

Index