Package ‘RcppML’

July 28, 2021

Type Package
Title Rcpp Machine Learning Library
Version 0.1.0
Date 2021-06-29
Description High-performance machine learning algorithms implemented in 'Rcpp' and 'RcppEigen', including non-negative factorization of sparse matrices, projection of linear factor models, and non-negative least squares.
License GPL (>= 2)
Imports Rcpp, Matrix, methods, stats
LinkingTo Rcpp, RcppEigen
VignetteBuilder knitr
RoxygenNote 7.1.1
Suggests knitr, rmarkdown, testthat (>= 3.0.0)
Config/testthat/edition 3
URL https://github.com/zdebruine/RcppML
BugReports https://github.com/zdebruine/RcppML/issues
NeedsCompilation yes
Author Zachary DeBruine [aut, cre] (<https://orcid.org/0000-0003-2234-4827>)
Maintainer Zachary DeBruine <zacharydebruine@gmail.com>
Repository CRAN
Date/Publication 2021-07-28 12:10:08 UTC

R topics documented:

 mse ................................................................. 2
 nmf ............................................................... 3
 nnls .............................................................. 6
 project ........................................................... 8
 RcppML .......................................................... 10
Mean Squared Error loss of a factor model

Description

MSE of factor models w and h given sparse matrix A

Usage

mse(A, w, d = NULL, h, threads = 0)

Arguments

A sparse matrix (of or coercible to dgCMatrix) of samples (columns) by features (rows)
w dense matrix of class matrix with factors (columns) by features (rows)
d optional diagonal scaling vector (if other than 1's) of rank length
h dense matrix of class matrix with samples (columns) by factors (rows)
threads number of CPU threads for parallelization, default 0 for all available threads.

Details

Calculates the cross-product of wh or wdh (if d is specified), subtracts that from A, squares the result, and calculates the mean of all values.

Parallelization is used across all available threads as determined by OpenMP.

Sparse matrix iterators are used in a C++ backend for high performance.

Value

scalar, MSE of the factorization

Author(s)

Zach DeBruine

Examples

library(Matrix)
A <- Matrix::rsparsematrix(1000, 1000, 0.1)
model <- nmf(A, k = 10)
mse(A, model$w, model$d, model$h)
Non-negative matrix factorization

Description
High-performance matrix factorization with optional non-negativity constraints and L1 regularization.

Usage
nmf(
  A,      
  k,      
  tol = 0.001,  
  maxit = 100,  
  verbose = TRUE,  
  nonneg = TRUE,  
  L1 = c(0, 0),  
  seed = NULL,  
  threads = 0,  
  ...  
)

Arguments
- **A**: sparse matrix of features x samples, of or coercible to class Matrix::dgCMatrix
- **k**: rank
- **tol**: correlation distance between \( w \) across consecutive iterations at which to stop factorization
- **maxit**: maximum number of alternating updates of \( w \) and \( h \)
- **verbose**: print tolerances after each iteration to the console
- **nonneg**: apply non-negativity constraints
- **L1**: L1/LASSO penalty, generally between 0 and 1, array of length two for \( c(w, h) \)
- **seed**: random seed for initializing \( w \) with C++ srand and RNG
- **threads**: number of CPU threads for parallelization, default 0 for all available threads

Details
This fast non-negative matrix factorization (NMF) implementation decomposes a sparse matrix \( A \) into orthogonal lower-rank non-negative matrices \( w \) and \( h \), with factors scaled to sum to 1 by a diagonal, \( d \):

\[
A = wdh
\]

For theoretical details, please see our manuscript: "DeBruine ZJ, Melcher K, Triche TJ (2021) High-performance non-negative matrix factorization for large single cell data." on BioRXiv.
Value

A list giving the factorization model:

- \( w \) : feature factor matrix
- \( d \) : scaling diagonal vector
- \( h \) : sample factor matrix

Stopping criteria

Use the \( \text{tol} \) parameter to control the stopping criteria for alternating updates:

- \( \text{tol} = 1e-2 \) is appropriate for approximate mean squared error determination and coarse cross-validation, for example in rank determination
- \( \text{tol} = 1e-3 \) to \( 1e-4 \) are suitable for rapid experimentation, cross-validation, and preliminary analysis
- \( \text{tol} = 1e-5 \) and smaller for publication-quality runs

The \( \text{maxit} \) parameter is a secondary stopping criterion that takes effect only if \( \text{tol} \) is not satisfied by the maximum number of specified iterations.

L1 regularization

L1 penalization introduces sparsity into all factors. Because RcppML NMF models are diagonalized, sparsity is enforced equally across all factors regardless of initialization and iteration. Use the L1 parameter to set any desired L1/LASSO penalty on the right-hand side of systems of equations during updates of \( w \) or \( h \). Typical values range between 0 and 1, where 1 is extremely or entirely sparse (this may lead to numerical issues).

Reproducibility

The optional \( \text{seed} \) parameter may be specified to guarantee absolute reproducibility between restarts. The seed is set on the R end. Note that only random initialization is supported, as other initializations do not show better performance and can trap the updates into local minima.

Because random initializations are used, the resulting model may be slightly different, especially for factors with lower diagonal weights. The same information is explained by all models, but in slightly different splits across less robust factors. Use of cross-validation can help optimize rank and L1 regularization to improve robustness if needed.

Rank determination

Like any clustering algorithm or dimensional reduction, finding the optimal rank can be a subjective process. An easy way to estimate rank that works for well-conditioned matrices uses the "elbow method", where the inflection point on a plot of Mean Squared Error loss (MSE) vs. rank gives a good idea of the rank at which most of the signal has been captured in the model. Unfortunately, this inflection point is not often as obvious for NMF as it is for SVD or PCA.

Better methods include cross-validation against robustness objectives, such as k-fold test-training splits. Missing value of imputation has previously been proposed, but is arguably no less subjective than test-training splits and requires computationally slower factorization updates.
Advanced parameters

Several parameters hidden in the ... argument may be adjusted (although defaults should entirely satisfy) in addition to those documented explicitly:

- **cd_maxit**, default 1000. Maximum number of coordinate descent iterations for solution refinement after initialization with solution from previous iteration. Only used as stopping criterion if **cd_tol** is not satisfied previously. See *nnls*.

- **fast_maxit**, default 10. Maximum number of FAST iterations for finding an approximate solution to initialize coordinate descent. See *nnls*.

- **cd_tol**, default 1e-8. Stopping criterion for coordinate descent iterations given by the maximum relative change in any coefficient between consecutive solutions. See *nnls*.

- **diag**, default TRUE. Enable model diagonalization to normalize factors to sum to 1, guarantee symmetry for symmetric factorizations, and distribute L1 penalties consistently and equally across all factors.

Author(s)

Zach DeBruine

References


See Also

*nnls, project, mse*

Examples

```r
library(Matrix)
# basic NMF
model <- nmf(rsparsematrix(1000, 100, 0.1), k = 10)

# symmetric NMF
A <- crossprod(rsparsematrix(100, 100, 0.02))
model <- nmf(A, 10, tol = 1e-5, maxit = 1000)
plot(model$w, t(model$h))
# see package vignette for more examples
```
nnls  Non-negative least squares

Description

Solves the equation \( a \times x = b \) for \( x \), where \( b \) can be either a vector or a matrix, subject to non-negativity constraints (if specified).

Usage

```
nls(
  a,
  b,
  x = NULL,
  nonneg = TRUE,
  fast_maxit = 10,
  cd_maxit = 1000,
  cd_tol = 1e-08
)
```

Arguments

- `a`: symmetric positive definite matrix giving coefficients of linear system
- `b`: vector or matrix giving right-hand side(s) of linear system
- `x`: initial value for \( x \) if using only coordinate descent least squares
- `nonneg`: enforce non-negativity of the solution
- `fast_maxit`: maximum number of FAST iterations for finding an approximate solution to initialize coordinate descent
- `cd_maxit`: maximum number of coordinate descent iterations for solution refinement after initialization (with either \( x \) or FAST if \( x = NULL \)). Only used as stopping criterion if \( cd\_tol \) is not satisfied previously.
- `cd_tol`: stopping criterion for coordinate descent iterations given by the maximum relative change in any coefficient between consecutive solutions

Details

This is a very fast implementation of non-negative least squares that outperforms Lawson-Hanson NNLS in our benchmarks for most applications, particularly for systems >25 variables in size.

If `nonneg = FALSE`, the Eigen Cholesky module is used to solve the system of equations, and is usually faster than `base::solve`

NNLS solutions will be found quickly using default parameters. If \( x \) is provided, only sequential coordinate descent least squares will be run. If \( x = NULL \), solutions will be initialized with "FAST" if `nonneg = TRUE` or solved using the Eigen Cholesky module if `nonneg = FALSE`. The FAST method also makes use of the Eigen Cholesky module.

If \( a \) is not positive, coordinate descent least squares will always be used, and with zero-filled initialization if \( x = NULL \).
Value

vector or matrix giving solution for x

Coordinate Descent NNLS

Least squares by sequential coordinate descent is used to ensure the solution returned is exact. This algorithm was introduced by Franc et al. (2005), and our implementation is adapted from the NNLM R package by Lin and Boutros (2020). There are two ways to initialize the coordinate descent solver:

1. Specify a value for x, either a random or zero-filled vector, or an approximate solution.
2. Leave x = NULL and use FAST initialization.

FAST NNLS

Forward active set tuning (FAST) is an exact or near-exact NNLS approximation initialized by an unconstrained least squares solution. Negative values in this unconstrained solution are set to zero (the "active set"), and all other values are added to a "feasible set". An unconstrained least squares solution is then solved for the "feasible set", any negative values in the resulting solution are set to zero, and the process is repeated until the feasible set solution is strictly positive.

The FAST algorithm has a definite convergence guarantee because the feasible set will either converge or become smaller with each iteration. The result is generally exact or nearly exact for small well-conditioned systems (< 50 variables) within 2 iterations and thus quickly sets up coordinate descent very well. The FAST method is similar to the first phase of the so-called "TNT-NN" algorithm (Myre et al., 2017), but the latter half of that method relies heavily on heuristics to find the true active set, which we avoid by using coordinate descent instead.

See our BioRXiv manuscript for benchmarking against Lawson-Hanson NNLS and for a more technical introduction to these methods.

Author(s)

Zach DeBruine

References


See Also

nmf, project
Examples

# compare solution to base::solve for a random system
X <- matrix(runif(100), 10, 10)
a <- crossprod(X)
b <- crossprod(X, runif(10))
unconstrained_soln <- solve(a, b)
nonneg_soln <- nnls(a, b)
unconstrained_err <- mean((a %*% unconstrained_soln - b)^2)
nonnegative_err <- mean((a %*% nonneg_soln - b)^2)
unconstrained_err
nonnegative_err
all.equal(solve(a, b), nnls(a, b, nonneg = FALSE))

# example adapted from multiway::fnnls example 1
X <- matrix(1:100, 50, 2)
y <- matrix(101:150, 50, 1)
beta <- solve(crossprod(X)) %*% crossprod(X, y)
beta
beta <- nnls(crossprod(X), crossprod(X, y))

# example adapted from multiway::fnnls example 4
X <- matrix(rnorm(2000), 100, 20)
btrue <- runif(20)
y <- X %*% btrue + rnorm(100)
beta <- nnls(crossprod(X), crossprod(X, y))
crossprod(btrue-beta)/20

---

project

*Project a linear factor model*

Description

Solves the equation \( A = wh \) for \( h \)

Usage

project(A, w, nonneg = TRUE, L1 = 0, threads = 0, ...)

Arguments

- **A**: sparse matrix of features x samples, of or coercible to class `Matrix::dgCMatrix`
- **w**: dense matrix of features x factors giving the linear model to be projected
- **nonneg**: apply non-negativity constraints
- **L1**: L1/LASSO penalty to be applied to \( h \). Generally should be scaled to \( \max(b) \) where \( b = WA \), for all columns \( j \) in \( A \)
- **threads**: number of CPU threads for parallelization, default 0 for all available threads.
- **...**: advanced parameters, see details
Details

For the classical alternating matrix factorization update problem \( A = w h \), the updates (or projection) of \( h \) is given by the equations:

\[
w^T w h = w A_j
\]

in the form \( a x = b \) where \( a = w^T w \) \( x = h \) and \( b = w A_j \) for all columns \( j \) in \( A \).

Given \( A \) and \( w \), RcppML::project solves for \( h \) using the above formulas and RcppML::nnls.

The corresponding equation for updating \( w \) in block-pivoting is:

\[
h h^T w^T = h A_j^T
\]

Thus, one may also solve for \( w \) by inputting the transpose of \( A \) and \( h \) in place of \( w \).

Value

matrix \( h \)

Advanced parameters

Several parameters hidden in the ... argument may be adjusted (although defaults should entirely satisfy) in addition to those documented explicitly:

- \texttt{cd\_maxit}, default 1000. Maximum number of coordinate descent iterations for solution refinement after initialization with solution from previous iteration. Only used as stopping criterion if \texttt{cd\_tol} is not satisfied previously. See \texttt{nnls}.
- \texttt{fast\_maxit}, default 10. Maximum number of FAST iterations for finding an approximate solution to initialize coordinate descent. See \texttt{nnls}.
- \texttt{cd\_tol}, default 1e-8. Stopping criterion for coordinate descent iterations given by the maximum relative change in any coefficient between consecutive solutions. See \texttt{nnls}.

Author(s)

Zach DeBruine

See Also

\texttt{nnls}, \texttt{nmf}

Examples

```r
library(Matrix)
w <- matrix(runif(1000 * 10), 1000, 10)
h_true <- matrix(runif(10 * 100), 10, 100)
A <- (w %*% h_true) * (rsparsematrix(1000, 100, 0.5) > 0)
h <- project(A, w)
cor(as.vector(h_true), as.vector(h))
# alternating projections refine solution (like NMF)
mse(A, w, h = h) # mse before alternating updates
h <- project(A, w)
```
RcppML

RcppML: Rcpp Machine Learning Library

Description

High-performance non-negative matrix factorization and linear model projection for sparse matrices, and fast non-negative least squares implementations

Author(s)

Zach DeBruine

See Also

Useful links:

- [https://github.com/zdebruine/RcppML](https://github.com/zdebruine/RcppML)
- Report bugs at [https://github.com/zdebruine/RcppML/issues](https://github.com/zdebruine/RcppML/issues)
Index

_PACKAGE (RcppML), 10

mse, 2, 5

nmf, 3, 7, 9
nnls, 5, 6, 9
project, 5, 7, 8

RcppML, 10
RcppML-package (RcppML), 10