libNMF 1.03

January 23, 2011

This document describes the functionality and outlines implementation steps for every major function in this library. For details on calling these functions see the according source files.

Contents

1 General notes

Throughout this library matrices are stored in one-dimensional arrays. To simplify the use of fortran high performance routines these arrays are logically accessed in column-major order, which is how fortran stores two-dimensional arrays.

The libnmf utilizes routines from BLAS, LAPACK and ARPACK. In the following lines I will shortly discuss our experiences with different versions of these libraries.

BLAS - we tested *ATLAS*- (version 3.8.3 and development version 3.9.11), *GOTO*-BLAS (version 1.26) and *GOTO2*-BLAS (version 1.13). Overall GOTO-BLAS seemed to be faster, setting an emphasis on parallel performance with no difference between GOTO and GOTO2. ATLAS-BLAS seemed to utilize cpu cores considerably less, which improved in the newer version (which features a new multithreading-implementation). There also seems to be an impact of BLAS versions on runtime stability per iteration (see *Known Issues*).

ARPACK - we tested *PROPACK* and *ARPACK* for calculation of the singular value decomposition in matrix generation by *NNDSVD*. While both ran fine for smaller sized problems, PROPACK had convergence issues with bigger Problems and we therefore only utilize ARPACK now.

Precision - we default to double precision but support single precision as well. To this intent every driver and computational routine exists in a version with an appended `_singleprec`. These single precision versions are therefore only mentioned for auxilliary routines.

1.1 Known Issues

The number of small positive (in the order of machine precision) and zero entries increase in the course of several iterations which increased the computational time per iteration in performance tests as well. Therefore small positive entries are set to zero in every iteration. The according threshold `ZERO_THRESHOLD` can be set in the header file `common.h`; if not set the machine precision (as returned by BLAS routine `DLAMCH`) is used by default. Additionaly in function `nmf_mu` the factor `DIV_BY_ZERO_AVOIDANCE` is defined with a default value of 10^{-9}. This factor is added to the divisor in an elementwise calculation to avoid division by zero and division by a very small number (close to machine precision). The latter case would result in a slowdown of iterations in the course of several iterations.

In function `nmf_mu` an increasing number of zero entries slowed down the computation as well. Performance tests showed that a relatively constant runtime per iteration could be achieved by zero-comparison of relevant entries prior to the computation; in case of a zero the result is then directly set to zero.

2 Driver routines

2.1 nmfDriver

Purpose The *nmfDriver* routine calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

It takes care of everything needed for several runs of factorizations, finally storing the best result in a file.

Used BLAS/LAPACK routines Not directly used

Implementation outline This routine first of checks the arguments passed in the call and sets default options if needed. It then loads a matrix A (format as stated in *loadMatrix* and *storeMatrix*) and optionally initial factor matrices W0 and H0 (if not loaded they get initialised with random numbers).

For every repetition the computational routine according to the chosen factorization method is called and the factor matrices get re-initialized afterwards. The best result - in terms of root mean square residual - is normalized and resorted according to the length of row vectors of matrix H.

Implemented algorithms to actually compute the factorization are (for details see each algorithm):

nmf_mu	multiplicative update approach
nmf_als	alternating least squares approach
nmf_neals	normal equation alternating least squares approach
nmf_alspg	alternating least squares using a projected gradient method
nmf_pg	direct projected gradient approach
nmf_fast	fast least-squares approach (unoptimized)
nmf_bayes	iterated conditional mode non-negative matrix factorization (unoptimized)

3 Computational routines

3.1 nmf_mu

Purpose This routine calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines The following routines are called

DGEMM	for matrix-matrix multiplications
DLAMCH	for calculationg machine precision

Implementation outline In every factorization step each new factor matrix gets calculated by three DGEMM matrix-matrix multiplications and one elementwise calculation in a simple for-loop.

Factor matrix H gets calculated first and in the following way:

$$H_{n+1} = \frac{H_n \cdot (W_n^t * A)}{(W_n^t * W_n) * H_n + eps}$$

Each Matrix-Matrix Multiplication ($*$) is calculated by a DGEMM call and all elementwise operations ($\cdot$, $+$ and the fraction) are calculated in a for-loop.

Factor matrix W is caculated according to following equation:

$$W_{n+1} = \frac{W_n \cdot (A * H_{n+1}^t)}{W_n * (H_{n+1} * H_{n+1}^t) + eps}$$

As a stopping criterion the root mean square residual and the maximum change in the factor matrices is then calculated in every iteration (see functions *calculateNorm* and *calculateMaxchange*).

3.2 nmf_als

Purpose This routine calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines The following routines are called

DGEQP3	for performing a QR-factorization with column pivoting
DORGQR	for generating an economy sized explicit Q
DGEMM	for matrix matrix multiplications
DTRTRS	for solving a triangular system
DLACPY	for copying a matrix to another matrix
DCOPY	for copying a vector to another vector
DLAMCH	for calculationg machine precision

Implementation outline Since the matrices A and W get overwritten during some of the calculations, at the beginning of every iteration step they get copied to help variables.

For calculation of H the following steps are performed:

- QR-factorization with column pivoting of W (m x k) (DGEQP3)

- Copying R (k x k) out of resulting matrix (DLACPY)

- Calculate economy sized explicit Q (m x k) (DORGQR)

- Calculate $Qh = Q^t * A$ (DGEMM)

- Solve $R * x = Qh$ (DTRTRS)

- Permutate result according to the pivoting of the factorization (DCOPY) and set negative entries to zero

Before the calculation of W begins, matrix A and H are copied and transposed to help variables and then the following steps are performed:

- QR-factorization with column pivoting of H^t (n x k) (DGEQP3)

- Copying R (k x k) out of resulting matrix (DLACPY)

- Calculate economy sized explicit Q (n x k) (DORGQR)

- Calculate $Qw = Q^t * A^t$ (DGEMM)

- Solve $R * x = Qw$ (DTRTRS)

- Permutate and transpose result according to the pivoting of the factorization (DCOPY) and set negative entries to zero

As a stopping criterion the root mean square residual and the maximum change in the factor matrices is then calculated in every iteration (see functions *calculateNorm* and *calculateMaxchange*).

3.3 nmf_neals

Purpose   This routine calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines   The following routines are called

DGEMM	for matrix-matrix multiplications
DGESV	for solving a system of lin. equations using a LU-factorization
DLAMCH	for calculationg machine precision

Implementation outline   For calculation of H the following steps are performed:

- Calculating of $help1 = W^t * W$ (DGEMM)

- Calculating of $help2 = W^t * A$ (DGEMM)

- Solving $help1 * x = help2$ using LU-factorization (DGESV)

- Set negative matrix entries to zero (loop)

For calculation of W the following steps are performed:

- Calculating of $help1 = H * H^t$ (DGEMM)

- Calculating of $help3 = H * A^t$ (DGEMM)

- Solving $help1 * x = help3$ using LU-factorization (DGESV)

- Transpose the result (DCOPY) and set negative matrix entries to zero (loop)

As a stopping criterion the root mean square residual and the maximum change in the factor matrices is then calculated in every iteration (see functions *calculateNorm* and *calculateMaxchange*).

Notes   While Matlab computes the condition when using a LU-factorization this code skips this step since there already is a convergence check every iteration.

3.4 nmf_alspg

Purpose   This routine [1] calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines   The following routines are called

DGEMM	for matrix-matrix multiplications
DCOPY	for transposing matrices
DLANGE	for calculating the frobenius norm

Implementation outline   Before the iterations begin the initial gradients get calculated as follows: (three times DGEMM each)

$$gradw = W * H * H^t - A * H^t$$

$$gradh = W^t * W * H - W^t * A$$

Additionally the initial gradient norm is calculated (DLANGE), wich serves as a stopping condition.

In every iteration first the new projection norm is calculated. (In Matlab-Notation: $norm([gradW(gradW < 0|W > 0); gradH(gradH < 0|H > 0)]))$

For calculation of W the following steps are performed:

- Transposing matrices W and H (DCOPY)

- Solving the subproblem for A^t, H^t, W^t (see *pg_subprob*)

- Transposing W and gradW

For calculation of H the following steps are performed:

- Solving the subproblem for A, W, H (see *pg_subprob*)

As the final step for every iteration the root mean square residual is calculated.

3.5 nmf_pg

Purpose   This routine [1] calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines The following routines are called

DGEMM	for matrix-matrix multiplications
DCOPY	for transposing matrices
DLANGE	for calculating the frobenius norm
DLACPY	for copying matrices
DAXPY	for calculating y = a*x + y

Implementation outline In every iteration first the initial gradients get calculated as follows: (three times DGEMM each)

$$gradw = W * H * H^t - A * H^t$$

$$gradh = W^t * W * H - W^t * A$$

In the first iteration the initial gradient (DLANGE), the subproblem for A, W, H (see *pg_subproblem*) and the root mean square residual are calculated.
Afterwards iteration two begins.

In every ohter iteration first the new projection norm is calculated. (In Matlab-Notation: $norm([gradW(gradW < 0|W > 0); gradH(gradH < 0|H > 0)])$) as a stopping criterion.

For calculation of W the following steps are performed:

- Copy W to a temporary matrix (DLACPY)

- Calculate $W_n = W_n - \alpha * gradW$ (DAXPY) where α is the stepsize

- Set negative elements to zero (loop)

For calculation of H the following steps are performed:

- Copy H to a temporary matrix (DLACPY)

10

- Calculate $H_n = H_n - \alpha * gradH$ (DAXPY) wheres α is the stepsize

- Set negative elements to zero (loop)

After these steps the search for the next stepsize begins with following actions (loop):

- Decide whether to increase or decrease α.

- Calculate new matrices W and H (see former calculation of W and H)

- Store the better one of the last two results.

As the final step for every iteration the root mean square residual is calculated.

3.6 pg_subprob_h

Purpose This routine [1] solves a projected gradient subproblem for calculation of the next iteration of H as required by *nmf_alspg* and *nmf_pg*

Used BLAS/LAPACK routines The following routines are called

DGEMM	for matrix-matrix multiplications
DLACPY	for copying matrices
DAXPY	for calculating y = a*x + y

Implementation outline First the help matrices

$$WTA = W^t * A$$

and

$$WTW = W^t * W$$

get calculated (DGEMM).

In every iteration then the new gradient $grad = WTW * H - WTA$ is calculated (DLACPY and DGEMM)

As a stopping criterion the projection norm is computed and an inner loop for determining a step size ist entered.

For determining the step size $H_n = max(H - \alpha * grad, 0)$ is calculated and α gets updated based on the change from current H to H_n. Depending on the update of α the next H is chosen from previous H_p and next H_n (DLACPY).

3.7 pg_subprob_w

Purpose   This routine [1] is a transposed version of routine *pg_subprob_h* for reducing computational time.

3.8 nmf_fast

Purpose   This routine [2] calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines   The following routines are called

DGEMM	for matrix matrix multiplications
DLACPY	for copying a matrix to another matrix
DSYTRF	for creating the bunch-kaufmann factorization
DSYTRS	for solving the least-square problem

Implementation outline   The implementation of this algorithm ist still unoptimized.

3.9 nmf_bayes

Purpose   This routine [2] calculates a non negative matrix factorisation of a m by n matrix A

$$A = W * H$$

where A, W and H are non-negative matrices.
W is a m by k matrix
H is a k by n matrix
k is the approximation factor

Used BLAS/LAPACK routines The following routines are called

DGEMM	for matrix matrix multiplications
DCOPY	for copying vectors
DAXPY	for combined multiplication and add

Implementation outline The implementation of this algorithm ist still unoptimized.

4 Auxiliary routines

4.1 calculateNorm

Purpose Calculates the frobenius norm of matrix $D = A - W * H$ which is then divided by $sqrt(m * n)$, where m and n are the matrix dimensions of A.

Used BLAS/LAPACK routines The following routines are called

DGEMM	for matrix-matrix multiplications
DLANGE	for computation of the frobenius norm
DLACPY	for copying a matrix

Implementation outline At first the matrix D ist computed (DLACPY and DGEMM). Then the frobenius norm of this matrix is calculated and divided by the squareroot of the number of its elements (DLANGE).

4.2 calculateNorm_singleprec

Purpose This routine is a single precision version of *calculateNorm* and uses single precision versions of BLAS and LAPACK routines.

4.3 calculateMaxchange

Purpose Calculates the maximum change matrix w or h show from last iteration to the current iteration, which is used as one of the convergence checks in most implemented nmf algorithms.

Used BLAS/LAPACK routines The following routines are called

DLANGE	for computation of the frobenius norm
DAXPY	for addition of matrices

Implementation outline First the absolute maximum of the previous matrix is computed (DLANGE). Then the difference of current and previous matrix is calculated (DAXPY).

The function then returns the ratio of absolute maximum of the difference matrix (DLANGE) and the absolute maximum of the previous matrix.

4.4 calculateMaxchange_singleprec

Purpose This routine is a single precision version of *calculateMaxchange* and uses single precision version of BLAS and LAPACK routines.

4.5 loadMatrix

Purpose Loads a matrix from the specified file and allocates memory according to the matrix dimensions

Used BLAS/LAPACK routines Not used.

Implementation outline The file has to be a simple ascii file where the first row has exactly one entry which is the number of rows m and the second row has exactly one entry which is the number of columns n. The next m rows contain the elements of the matrix in row-major-order where every element is separated by a single space.

4.6 loadMatrix_singleprec

Purpose This routine is a single precision version of *loadMatrix* and uses single precision versions of BLAS and LAPACK routines.

4.7 storeMatrix

Purpose Stores a matrix from the specified file and allocates memory according to the matrix dimensions.

Used BLAS/LAPACK routines Not used.

Implementation outline The file will be a simple ascii file where the first row has exactly one entry which is the number of rows m and the second row has exactly one entry which is the number of columns n. The next m rows contain the elements of the matrix in row-major-order where every element is separated by a single space.

4.8 storeMatrix_singleprec

Purpose   This routine is a single precision version of *storeMatrix* and uses single precision versions of BLAS and LAPACK routines.

4.9 generateMatrix

Purpose   Allocates memory for a matrix and initializes this matrix using one of the following methods

- random values (`init_rand`)

- non negative double SVD (`init_nndsvd`)

- information gain (`init_infogain`) (requires classification data)

- gain ratio (`init_gainratio`) (requires classification data)

Used BLAS/LAPACK routines   Not used.

Implementation outline   The memory will be allocated as a one-dimensional array and logically used in column-major order. Every element will then be set to a random value.

4.10 generateMatrix_singleprec

Purpose   This routine is a single precision version of *generateMatrix* and uses single precision versions of BLAS and LAPACK routines.

4.11 setdefaultopts

Purpose   If a null-pointer is passed to nmfDriver for the options-structure default values are used wich this routine implements.

Used BLAS/LAPACK routines   Not used.

Implementation outline   The options structure contains following elements:

element	description	default v
rep	number of repetitions	1
init	method to use for initialising matrices	ran
min_init	minimal value for random numbers	0
max_init	maximal value for random numbers	1
w_out	filename to store final matrix w in	"final_w.m
h_out	filename to store final matrix h in	"final_h.m
TolX	tolerance limit for maxChange	1E-0
TolFun	tolerance for root mean square residual	1E-0
nndsvd_maxiter	maximum iterations for SVD in ddsvd initialisation	
nndsvd_blocksize	blocksize for SVD in ddsvd initialisation	
nndsvd_tol	tolerance for SVD in ddsvd initialisation	
nndsvd_ncv	largest number of basis vectors in the Implicitly Restarted Arnoldi Process	
num_dist_classes	count of distinct class values	
*classColumn	pointer to vector of class attributes for given Matrix A	
parallelization	Method to use for parallelization: par_pthreads, par_openmp	

4.12 setdefaultopts_singleprec

Purpose   This routine is a single precision version of *setdefaultopts* and uses single
precision versions of BLAS and LAPACK routines.

4.13 calculateSVD

Purpose   Calculates the SVD of a Matrix A.

Used BLAS/LAPACK routines	DSAUPD	for calculating the Singular Values
	DSEUPD	for calculating the Singular Vectors
	DGEMV	for Matrix-Vector fights
	DSCAL	for scaling of vectors
	DCOPY	for copying of vectors

4.14 calculateSVD_singleprec

Purpose   This routine is a single precision version of *calculateSVD* and uses single
precision versions of BLAS and LAPACK routines.

4.15 infogain

Purpose   Calculate the information gain and gainratio indices for an instance-times-
attribute matrix A.

Used BLAS/LAPACK routines   Not directly called.

Implementation outline This version uses POSIX Threads to parallelize the information gain calculations across columns.

4.16 infogain_singleprec

Purpose This routine is a single precision version of *infogain* and uses single precision versions of BLAS and LAPACK routines.

4.17 infogain_openmp

Purpose Calculate the information gain and gainratio indices for an instance-times-attribute matrix A.

Used BLAS/LAPACK routines Not directly called.

Implementation outline This version uses OpenMP to parallelize the information gain calculations across columns.

4.18 infogain_openmp_singleprec

Purpose This routine is a single precision version of *infogain_openmp* and uses single precision versions of BLAS and LAPACK routines.

References

[1] Chih-Jen Lin. Projected gradient methods for nonnegative matrix factorization. *Neural Comput.*, 19(10):2756–2779, 2007.

[2] Mikkel N. Schmidt and Hans Laurberg. Non-negative matrix factorization with Gaussian process priors. *Comp. Intelligence and Neuroscience*, 2008(1):1–10, 2008.