
SPAMS Documentation

Release 1

jp

July 04, 2014

CONTENTS

1	sort	3
2	calcAAAt	5
3	calcXAt	7
4	calcXY	9
5	calcXYt	11
6	calcXtY	13
7	bayer	15
8	conjGrad	17
9	invSym	19
10	normalize	21
11	sparseProject	23
12	lasso	25
13	lassoMask	27
14	lassoWeighted	29
15	omp	31
16	ompMask	33
17	cd	35
18	somp	37
19	l1L2BCD	39
20	fistaFlat	41
21	fistaTree	45
22	fistaGraph	49

23	proximalFlat	53
24	proximalTree	57
25	proximalGraph	61
26	trainDL	63
27	structTrainDL	67
28	trainDL_Memory	71
29	nmf	75
30	nnsc	77
31	archetypalAnalysis	79
32	decompSimplex	81
33	simpleGroupTree	83
34	readGroupStruct	85
35	groupStructOfString	87
36	graphOfGroupStruct	89
37	treeOfGroupStruct	91
38	Indices and tables	93
	Index	95

Contents:

<code>sort(X[, mode])</code>	sort the elements of X using quicksort
<code>calcAAAt(A)</code>	Compute efficiently $AA^T = A^*A'$, when A is sparse and has a lot more columns than rows
<code>calcXAt(X, A)</code>	Compute efficiently $XA^T = X^*A'$, when A is sparse and has a lot more columns than rows
<code>calcXY(X, Y)</code>	Compute $Z=XY$ using the BLAS library used by SPAMS.
<code>calcXYt(X, Y)</code>	Compute $Z=XY'$ using the BLAS library used by SPAMS.
<code>calcXtY(X, Y)</code>	Compute $Z=X'Y$ using the BLAS library used by SPAMS.
<code>bayer(X, offset)</code>	bayer applies a Bayer pattern to an image X.
<code>conjGrad(A, b[, x0, tol, itermax])</code>	Conjugate gradient algorithm, sometimes faster than the equivalent python implementation
<code>invSym(A)</code>	returns the inverse of a symmetric matrix A
<code>normalize(A)</code>	rescale the columns of X so that they have unit norm
<code>sparseProject(U[, thrs, mode, lambda1, ...])</code>	sparseProject solves various optimization problems
<code>lasso(X[, D, Q, q, return_reg_path, L, ...])</code>	lasso is an efficient implementation of the l1-regularized least squares
<code>lassoMask(X, D, B[, L, lambda1, lambda2, ...])</code>	lasso is a variant of lasso that handles multiple data sets
<code>lassoWeighted(X, D, W[, L, lambda1, mode, ...])</code>	lassoWeighted is an efficient implementation of the weighted l1-regularized least squares
<code>omp(X, D[, L, eps, lambda1, ...])</code>	omp is an efficient implementation of the orthogonal matching pursuit
<code>ompMask(X, D, B[, L, eps, lambda1, ...])</code>	ompMask is a variant of omp that allow using multiple data sets
<code>cd(X, D, A0[, lambda1, mode, itermax, tol, ...])</code>	cd addresses l1-decomposition problem with a coordinate descent
<code>somp(X, D, list_groups[, L, eps, numThreads])</code>	somp is an efficient implementation of a sparse orthogonal matching pursuit
<code>l1L2BCD(X, D, alpha0, list_groups[, ...])</code>	l1L2BCD is a solver for a l1-l2-bcd problem
<code>fistaFlat(Y, X, W0[, return_optim_info, ...])</code>	fistaFlat solves sparse regularized problems.
<code>fistaTree(Y, X, W0, tree[, ...])</code>	fistaTree solves sparse regularized problems.
<code>fistaGraph(Y, X, W0, graph[, ...])</code>	fistaGraph solves sparse regularized problems.
<code>proximalFlat(U[, return_val_loss, ...])</code>	proximalFlat computes proximal operators. Depending on the mode, it can be used for l1, l2, or l1-l2
<code>proximalTree(U, tree[, return_val_loss, ...])</code>	proximalTree computes a proximal operator. Depending on the mode, it can be used for l1, l2, or l1-l2
<code>proximalGraph(U, graph[, return_val_loss, ...])</code>	proximalGraph computes a proximal operator. Depending on the mode, it can be used for l1, l2, or l1-l2
<code>trainDL(X[, return_model, model, D, ...])</code>	trainDL is an efficient implementation of the least squares
<code>structTrainDL(X[, return_model, model, D, ...])</code>	structTrainDL is an efficient implementation of the least squares
<code>trainDL_Memory(X[, D, numThreads, ...])</code>	trainDL_Memory is an efficient but memory consuming implementation of the least squares
<code>nmf(X[, return_lasso, model, numThreads, ...])</code>	trainDL is an efficient implementation of the non-negative matrix factorization
<code>nnsr(X[, return_lasso, model, lambda1, ...])</code>	trainDL is an efficient implementation of the non-negative matrix factorization
<code>archetypalAnalysis(X[, p, Z0, returnAB, ...])</code>	documentation to appear soon
<code>decompSimplex(X, Z[, computeXtX, numThreads])</code>	documentation to appear soon
<code>simpleGroupTree(degrees)</code>	makes a structure representing a tree given the degrees
<code>readGroupStruct(file)</code>	reads a text file describing “simply” the structure of groups
<code>groupStructOfString(s)</code>	decode a multi-line string describing “simply” the structure of groups
<code>graphOfGroupStruct(gstruct)</code>	converts a group structure into the graph structure
<code>treeOfGroupStruct(gstruct)</code>	converts a group structure into the tree structure

SORT

`spams.sort(X, mode=True)`
sort the elements of X using quicksort

Parameters

- X – double vector of size n
- **mode** – false for decreasing order (true by default)

Returns

- Y : double vector of size n

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

CALCAAT

`spams.calcAAt` (*A*)

Compute efficiently $\mathbf{AAt} = \mathbf{A} \cdot \mathbf{A}'$, when \mathbf{A} is sparse and has a lot more columns than rows. In some cases, it is up to 20 times faster than the equivalent python expression $\mathbf{AAt} = \mathbf{A} \cdot \mathbf{A}'$;

Parameters \mathbf{A} – double sparse $m \times n$ matrix

Returns

- **AAt**: double $m \times m$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

CALCXAT

`spams.calcXAt` (X, A)

Compute efficiently $XAt = X \cdot A'$, when A is sparse and has a lot more columns than rows. In some cases, it is up to 20 times faster than the equivalent python expression;

Parameters

- X – double $m \times n$ matrix
- A – double sparse $p \times n$ matrix

Returns

- XAt : double $m \times p$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

CALCXY

`spams.calcXY(X, Y)`

Compute $Z=XY$ using the BLAS library used by SPAMS.

Parameters

- **X** – double $m \times n$ matrix
- **Y** – double $n \times p$ matrix

Returns

- **Z**: double $m \times p$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

CALCXYT

`spams.calcXYt` (X, Y)

Compute $Z=XY'$ using the BLAS library used by SPAMS.

Parameters

- \mathbf{X} – double $m \times n$ matrix
- \mathbf{Y} – double $p \times n$ matrix

Returns

- \mathbf{Z} : double $m \times p$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

CALCXTY

`spams.calcXtY(X, Y)`

Compute $Z=X'Y$ using the BLAS library used by SPAMS.

Parameters

- **X** – double $n \times m$ matrix
- **Y** – double $n \times p$ matrix

Returns

- **Z**: double $m \times p$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

BAYER

spams.**bayer** (X , *offset*)

bayer applies a Bayer pattern to an image **X**. There are four possible offsets.

Parameters

- **X** – double $m \times n$ matrix
- **offset** – scalar, 0,1,2 or 3

Returns

- **Y**: double $m \times m$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

CONJGRAD

`spams.conjGrad(A, b, x0=None, tol=1e-10, itermax=None)`

Conjugate gradient algorithm, sometimes faster than the equivalent python function `solve`. In order to solve $Ax=b$;

Parameters

- **A** – double square $n \times n$ matrix. HAS TO BE POSITIVE DEFINITE
- **b** – double vector of length n .
- **x0** – double vector of length n . (optional) initial guess.
- **tol** – (optional) tolerance.
- **itermax** – (optional) maximum number of iterations.

Returns

- **x**: double vector of length n .

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

INVSYM

`spams.invSym(A)`

returns the inverse of a symmetric matrix A

Parameters A – double $n \times n$ matrix

Returns

- B : double $n \times n$ matrix

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

NORMALIZE

`spams.normalize(A)`

rescale the columns of X so that they have unit l2-norm.

Parameters **X** – double m x n matrix

Returns

- **Y**: double m x n matrix

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

SPARSEPROJECT

`spams.sparseProject(U, thrs=1.0, mode=1, lambda1=0.0, lambda2=0.0, lambda3=0.0, pos=0, numThreads=-1)`

`sparseProject` solves various optimization problems, including projections on a few convex sets.

It aims at addressing the following problems for all columns u of U in parallel

1. **when mode=1 (projection on the l1-ball)** $\min_v \|u-v\|_2^2$ s.t. $\|v\|_1 \leq \text{thrs}$

2. **when mode=2** $\min_v \|u-v\|_2^2$ s.t. $\|v\|_2^2 + \lambda \text{muda1} \|v\|_1 \leq \text{thrs}$

3. **when mode=3** $\min_v \|u-v\|_2^2$ s.t. $\|v\|_1 + 0.5 \lambda \text{muda1} \|v\|_2^2 \leq \text{thrs}$

4. **when mode=4** $\min_v 0.5 \|u-v\|_2^2 + \lambda \text{muda1} \|v\|_1$ s.t. $\|v\|_2^2 \leq \text{thrs}$

5. **when mode=5**

$$\min_v 0.5 \|u-v\|_2^2 + \lambda \text{muda1} \|v\|_1 + \lambda \text{muda2} \text{FL}(v) + \dots \quad 0.5 \lambda \text{muda3} \|v\|_2^2$$

where FL denotes a “fused lasso” regularization term.

6. **when mode=6** $\min_v \|u-v\|_2^2$ s.t. $\lambda \text{muda1} \|v\|_1 + \lambda \text{muda2} \text{FL}(v) + \dots$

$$0.5 \lambda \text{muda3} \|v\|_2^2 \leq \text{thrs}$$

When `pos=true` and `mode <= 4`, it solves the previous problems with positivity constraints

Parameters

- **U** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to project
- **thrs** – (parameter)
- **lambda1** – (parameter)
- **lambda2** – (parameter)
- **lambda3** – (parameter)
- **mode** – (see above)
- **pos** – (optional, false by default)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **V**: double $m \times n$ matrix (output matrix)

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting

LASSO

`spams.lasso(X, D=None, Q=None, q=None, return_reg_path=False, L=-1, lambda1=None, lambda2=0.0, mode=2, pos=False, ols=False, numThreads=-1, max_length_path=-1, verbose=False, cholesky=False)`

lasso is an efficient implementation of the homotopy-LARS algorithm for solving the Lasso.

If the function is called this way `spams.lasso(X,D = D, Q = None,...)`, it aims at addressing the following problems for all columns x of X , it computes one column α of A that solves

1. when `mode=0`

$$\min_{\{\alpha\}} \|x - D\alpha\|_2^2 \text{ s.t. } \|\alpha\|_1 \leq \lambda$$

2. when `mode=1`

$$\min_{\{\alpha\}} \|\alpha\|_1 \text{ s.t. } \|x - D\alpha\|_2^2 \leq \lambda$$

3. when `mode=2`

$$\min_{\{\alpha\}} 0.5\|x - D\alpha\|_2^2 + \lambda \|\alpha\|_1 + 0.5 \lambda^2$$

If the function is called this way `spams.lasso(X,D = None, Q = Q, q = q,...)`, it solves the above optimisation problem, when $Q=D'D$ and $q=D'x$.

Possibly, when `pos=true`, it solves the previous problems with positivity constraints on the vectors α

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary
- **Q** – $p \times p$ double matrix ($Q = D'D$)
- **q** – $p \times n$ double matrix ($q = D'X$)
- **verbose** – verbose mode
- **return_reg_path** – if true the function will return a tuple of matrices.
- **lambda1** – (parameter)
- **lambda2** – (optional parameter for solving the Elastic-Net) for `mode=0` and `mode=1`, it adds a ridge on the Gram Matrix

- **L** – (optional), maximum number of steps of the homotopy algorithm (can be used as a stopping criterion)
- **pos** – (optional, adds non-negativity constraints on the coefficients, false by default)
- **mode** – (see above, by default: 2)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **cholesky** – (optional, default false), choose between Cholesky implementation or one based on the matrix inversion Lemma
- **ols** – (optional, default false), perform an orthogonal projection before returning the solution.
- **max_length_path** – (optional) maximum length of the path, by default 4*p

Returns

- **A**: double sparse p x n matrix (output coefficients)
- **path**: optional, returns the regularisation path for the first signal $A = \text{spams.lasso}(X, \text{return_reg_path} = \text{False}, \dots)$ ($A, \text{path} = \text{spams.lasso}(X, \text{return_reg_path} = \text{True}, \dots)$)

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output alpha is double precision)

Examples:

```
import numpy as np
m = 5; n = 10; nD = 5
np.random.seed(0)
X = np.asfortranarray(np.random.normal(size=(m,n)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)), (X.shape[0],1)))
D = np.asfortranarray(np.random.normal(size=(100,200)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)), (D.shape[0],1)))
alpha = spams.lasso(X,D = D,return_reg_path = FALSE,lambda1 = 0.15)
```

LASSOMASK

spams.**lassoMask** (*X, D, B, L=-1, lambda1=None, lambda2=0.0, mode=2, pos=False, numThreads=-1, verbose=False*)

lasso is a variant of lasso that handles binary masks. It aims at addressing the following problems for all columns x of X , and β of B , it computes one column α of A that solves

1. when $\text{mode}=0$

$$\min_{\alpha} \{ \|\text{diag}(\beta)(x - D\alpha)\|_2^2 \text{ s.t. } \|\alpha\|_1 \leq \lambda_1$$

2. when $\text{mode}=1$

$$\min_{\alpha} \{ \|\alpha\|_1 \text{ s.t. } \|\text{diag}(\beta)(x - D\alpha)\|_2^2 \leq \lambda_1 * (\|\beta\|_0 / m)$$

3. when $\text{mode}=2$

$$\min_{\alpha} \{ \alpha \mid 0.5 \|\text{diag}(\beta)(x - D\alpha)\|_2^2 + \lambda_1 * (\|\beta\|_0 / m) * \|\alpha\|_1 + (\lambda_2 / 2) \|\alpha\|_2^2$$

Possibly, when $\text{pos}=\text{true}$, it solves the previous problems with positivity constraints on the vectors α

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary
- **B** – boolean $m \times n$ matrix (mask) p is the number of elements in the dictionary
- **verbose** – verbose mode
- **lambda1** – (parameter)
- **L** – (optional, maximum number of elements of each decomposition)
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)
- **mode** – (see above, by default: 2)
- **lambda2** – (optional parameter for solving the Elastic-Net) for $\text{mode}=0$ and $\text{mode}=1$, it adds a ridge on the Gram Matrix
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **A**: double sparse $p \times n$ matrix (output coefficients)

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output alpha is double precision)

LASSOWEIGHTED

`spams.lassoWeighted(X, D, W, L=-1, lambda1=None, mode=2, pos=False, numThreads=-1, verbose=False)`

`lassoWeighted` is an efficient implementation of the LARS algorithm for solving the weighted Lasso. It is optimized for solving a large number of small or medium-sized decomposition problem (and not for a single large one).

It first computes the Gram matrix $D'D$ and then perform a Cholesky-based OMP of the input signals in parallel. For all columns x of X , and w of W , it computes one column α of A which is the solution of

1. when `mode=0`

$$\min_{\{\alpha\}} \|x - D\alpha\|_2^2 \text{ s.t. } \|\text{diag}(w)\alpha\|_1 \leq \lambda$$

2. when `mode=1`

$$\min_{\{\alpha\}} \|\text{diag}(w)\alpha\|_1 \text{ s.t. } \|x - D\alpha\|_2^2 \leq \lambda$$

3. when `mode=2`

$$\min_{\{\alpha\}} 0.5\|x - D\alpha\|_2^2 + \lambda \|\text{diag}(w)\alpha\|_1$$

Possibly, when `pos=true`, it solves the previous problems with positivity constraints on the vectors α

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary
- **W** – double $p \times n$ matrix (weights)
- **verbose** – verbose mode
- **lambda1** – (parameter)
- **L** – (optional, maximum number of elements of each decomposition)
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)
- **mode** – (see above, by default: 2)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **A**: double sparse $p \times n$ matrix (output coefficients)

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output α is double precision)

OMP

`spams.omp(X, D, L=None, eps=None, lambda1=None, return_reg_path=False, numThreads=-1)`

`omp` is an efficient implementation of the Orthogonal Matching Pursuit algorithm. It is optimized for solving a large number of small or medium-sized decomposition problem (and not for a single large one).

It first computes the Gram matrix $D'D$ and then perform a Cholesky-based OMP of the input signals in parallel. $X=[x^1, \dots, x^n]$ is a matrix of signals, and it returns a matrix $A=[\alpha^1, \dots, \alpha^n]$ of coefficients.

it addresses for all columns x of X , $\min_{\alpha} \|x - D\alpha\|_2^2 \leq \text{eps}$ or $\min_{\alpha} \|x - D\alpha\|_2^2 \leq L$ or $\min_{\alpha} 0.5\|x - D\alpha\|_2^2 + \lambda_1 \|\alpha\|_1$

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary All the columns of D should have unit-norm !
- **`return_reg_path`** – if true the function will return a tuple of matrices.
- **L** – (optional, maximum number of elements in each decomposition, $\min(m, p)$ by default)
- **`eps`** – (optional, threshold on the squared l_2 -norm of the residual, 0 by default)
- **`lambda1`** – (optional, penalty parameter, 0 by default)
- **`numThreads`** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **A** : double sparse $p \times n$ matrix (output coefficients) `path` (optional): double dense $p \times L$ matrix (regularization path of the first signal) $A = \text{spams.omp}(X, D, L, \text{eps}, \text{return_reg_path} = \text{False}, \dots)$ (A, path) = $\text{spams.omp}(X, D, L, \text{eps}, \text{return_reg_path} = \text{True}, \dots)$

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output α is double precision) - Passing an int32 vector of length n to L provides a different parameter L for each input signal x_i - Passing a double vector of length n to `eps` and or `lambda1` provides a different parameter `eps` (or `lambda1`) for each input signal x_i

OMPMASK

`spams.ompMask(X, D, B, L=None, eps=None, lambda1=None, return_reg_path=False, numThreads=-1)`
ompMask is a variant of omp that allow using a binary mask B

for all columns x of X , and columns β of B , it computes a column α of A by addressing
$$\min_{\alpha} \{ \|\alpha\|_0 \text{ s.t. } \|\text{diag}(\beta)(x - D\alpha)\|_2^2 \leq \epsilon \|\beta\|_0 / m$$

or
$$\min_{\alpha} \{ \|\text{diag}(\beta)(x - D\alpha)\|_2^2 \text{ s.t. } \|\alpha\|_0 \leq L \text{ or } \min_{\alpha} \{ 0.5 \|\text{diag}(\beta)(x - D\alpha)\|_2^2 + \lambda_1 \|\alpha\|_0$$

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary All the columns of D should have unit-norm !
- **B** – boolean $m \times n$ matrix (mask) p is the number of elements in the dictionary
- **return_reg_path** – if true the function will return a tuple of matrices.
- **L** – (optional, maximum number of elements in each decomposition, $\min(m, p)$ by default)
- **eps** – (optional, threshold on the squared ℓ_2 -norm of the residual, 0 by default)
- **lambda1** – (optional, penalty parameter, 0 by default)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **A**: double sparse $p \times n$ matrix (output coefficients) **path** (optional): double dense $p \times L$ matrix (regularization path of the first signal) $A = \text{spams.ompMask}(X, D, B, L, \text{eps}, \text{return_reg_path} = \text{False}, \dots)$ (A, path) = $\text{spams.ompMask}(X, D, B, L, \text{eps}, \text{return_reg_path} = \text{True}, \dots)$

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output α is double precision) - Passing an int32 vector of length n to L provides a different parameter L for each input signal x_i - Passing a double vector of length n to eps and or lambda1 provides a different parameter eps (or lambda1) for each input signal x_i

`spams.cd(X, D, A0, lambda1=None, mode=2, itermax=100, tol=0.001, numThreads=-1)`

`cd` addresses l1-decomposition problem with a coordinate descent type of approach.

It is optimized for solving a large number of small or medium-sized decomposition problem (and not for a single large one). It first computes the Gram matrix $D'D$. This method is particularly well adapted when there is low correlation between the dictionary elements and when one can benefit from a warm restart. It aims at addressing the two following problems for all columns x of X , it computes a column α of A such that

2. when $\text{mode}=1$

$\min_{\alpha} \{ \|\alpha\|_1 \text{ s.t. } \|x - D\alpha\|_2^2 \leq \lambda \}$ For this constraint setting, the method solves a sequence of penalized problems (corresponding to $\text{mode}=2$) and looks for the corresponding Lagrange multiplier with a simple but efficient heuristic.

3. when $\text{mode}=2$

$\min_{\alpha} \{ 0.5\|x - D\alpha\|_2^2 + \lambda \|\alpha\|_1 \}$

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary All the columns of D should have unit-norm !
- **A0** – double sparse $p \times n$ matrix (initial guess)
- **lambda1** – (parameter)
- **mode** – (optional, see above, by default 2)
- **itermax** – (maximum number of iterations)
- **tol** – (tolerance parameter)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **A**: double sparse $p \times n$ matrix (output coefficients)

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)

- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output alpha is double precision)

SOMP

`spams.somp(X, D, list_groups, L=None, eps=0.0, numThreads=-1)`

somp is an efficient implementation of a Simultaneous Orthogonal Matching Pursuit algorithm. It is optimized for solving a large number of small or medium-sized decomposition problem (and not for a single large one).

It first computes the Gram matrix $D'D$ and then perform a Cholesky-based OMP of the input signals in parallel. It aims at addressing the following NP-hard problem

X is a matrix structured in groups of signals, which we denote by $X=[X_1, \dots, X_n]$

for all matrices X_i of X , $\min_{\|A_i\|_2} \|X_i - D A_i\|_2^2 \leq \text{eps} \cdot n_i$ where n_i is the number of columns of X_i

or

$\min_{\|A_i\|_2} \|X_i - D A_i\|_2^2 \text{ s.t. } \|A_i\|_2 \leq L$

Parameters

- **X** – double $m \times N$ matrix (input signals) m is the signal size N is the total number of signals
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary All the columns of D should have unit-norm !
- **list_groups** – int32 vector containing the indices (starting at 0) of the first elements of each groups.
- **L** – (maximum number of elements in each decomposition)
- **eps** – (threshold on the squared l2-norm of the residual)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **alpha**: double sparse $p \times N$ matrix (output coefficients)

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output alpha is double precision)

L1L2BCD

`spams.l1l2bcd(X, D, alpha0, list_groups, lambda1=None, mode=2, itermax=100, tol=0.001, numThreads=-1)`

l1l2bcd is a solver for a Simultaneous signal decomposition formulation based on block coordinate descent.

X is a matrix structured in groups of signals, which we denote by $X=[X_1, \dots, X_n]$

if mode=2, it solves for all matrices X_i of X , $\min_{\{A_i\}} 0.5\|X_i - D A_{i1}\|_2^2 + \lambda_1/\sqrt{n_i}\|A_{i1}\|_{1,2}$ where n_i is the number of columns of X_i

if mode=1, it solves $\min_{\{A_i\}} \|A_{i1}\|_{1,2}$ s.t. $\|X_i - D A_{i1}\|_2^2 \leq n_i \lambda_1$

Parameters

- **X** – double $m \times N$ matrix (input signals) m is the signal size N is the total number of signals
- **D** – double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary
- **alpha0** – double dense $p \times N$ matrix (initial solution)
- **list_groups** – int32 vector containing the indices (starting at 0) of the first elements of each groups.
- **lambda1** – (regularization parameter)
- **mode** – (see above, by default 2)
- **itermax** – (maximum number of iterations, by default 100)
- **tol** – (tolerance parameter, by default 0.001)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).

Returns

- **alpha**: double sparse $p \times N$ matrix (output coefficients)

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output alpha is double precision)

FISTAFLAT

```
spams.fistaFlat(Y, X, W0, return_optim_info=False, numThreads=-1, max_it=1000, L0=1.0,
fixed_step=False, gamma=1.5, lambda1=1.0, delta=1.0, lambda2=0.0, lambda3=0.0,
a=1.0, b=0.0, c=1.0, tol=1e-06, it0=100, max_iter_backtracking=1000, compute_gram=False,
lin_admm=False, admm=False, intercept=False, resetflow=False, regul='', loss='', verbose=False,
pos=False, clever=False, log=False, ista=False, subgrad=False, logName='', is_inner_weights=False,
inner_weights=None, size_group=1, groups=None, sqrt_step=True, transpose=False, linesearch_mode=0)
```

fistaFlat solves sparse regularized problems.

X is a design matrix of size $m \times p$ $X=[x^1, \dots, x^n]^T$, where the x_i 's are the rows of X $Y=[y^1, \dots, y^n]$ is a matrix of size $m \times n$ It implements the algorithms FISTA, ISTA and subgradient descent.

- if $\text{loss}='square'$ and regul is a regularization function for vectors, the entries of Y are real-valued, $W=[w^1, \dots, w^n]$ is a matrix of size $p \times n$ For all column y of Y , it computes a column w of W such that

$$w = \operatorname{argmin} 0.5 \|y - X w\|_2^2 + \lambda \psi(w)$$

- if $\text{loss}='square'$ and regul is a regularization function for matrices the entries of Y are real-valued, W is a matrix of size $p \times n$. It computes the matrix W such that

$$W = \operatorname{argmin} 0.5 \|Y - X W\|_F^2 + \lambda \psi(W)$$

- $\text{loss}='square-missing'$ same as $\text{loss}='square'$, but handles missing data represented by NaN (not a number) in the matrix Y

- if $\text{loss}='logistic'$ and regul is a regularization function for vectors, the entries of Y are either -1 or +1, $W=[w^1, \dots, w^n]$ is a matrix of size $p \times n$ For all column y of Y , it computes a column w of W such that

$$w = \operatorname{argmin} (1/m) \sum_{j=1}^m \log(1 + e^{-(y_j x^j w)}) + \lambda \psi(w),$$

where x^j is the j -th row of X .

- if $\text{loss}='logistic'$ and regul is a regularization function for matrices the entries of Y are either -1 or +1, W is a matrix of size $p \times n$

$$W = \operatorname{argmin} \sum_{i=1}^n (1/m) \sum_{j=1}^m \log(1 + e^{-(y^i_j x^j w^i)}) + \lambda \psi(W)$$

- if $\text{loss}='multi-logistic'$ and regul is a regularization function for vectors, the entries of Y are in $\{0, 1, \dots, N\}$ where N is the total number of classes $W=[W^1, \dots, W^n]$ is a matrix of size $p \times Nn$, each submatrix W^i is of size $p \times N$ for all submatrix WW of W , and column y of Y , it computes

$$WW = \operatorname{argmin} (1/m) \sum_{j=1}^m \log(\sum_{i=1}^N e^{(x^j w^i - ww^i y_j)}) + \lambda \sum_{j=1}^N \psi(ww^j),$$

where ww^j is the j -th column of WW .

- if `loss='multi-logistic'` and `regul` is a regularization function for matrices, the entries of Y are in $\{0,1,\dots,N\}$ where N is the total number of classes W is a matrix of size $p \times N$, it computes

$$W = \operatorname{argmin} (1/m) \sum_{j=1}^m \log(\sum_{j=1}^r e^{(x^j)'(w^j - w^{\{y_j\}})}) + \lambda \operatorname{psi}(W)$$

where w^j is the j -th column of W .

- `loss='cur'` useful to perform sparse CUR matrix decompositions**, $W = \operatorname{argmin} 0.5\|Y - X*W*X\|_F^2 + \lambda \operatorname{psi}(W)$

The function `psi` are those used by `proximalFlat` (see documentation)

This function can also handle intercepts (last row of W is not regularized), and/or non-negativity constraints on W , and sparse matrices for X

Parameters

- **`Y`** – double dense $m \times n$ matrix
- **`X`** – double dense or sparse $m \times p$ matrix
- **`W0`** – double dense $p \times n$ matrix or $p \times N_n$ matrix (for multi-logistic loss) initial guess
- **`return_optim_info`** – if true the function will return a tuple of matrices.
- **`loss`** – (choice of loss, see above)
- **`regul`** – (choice of regularization, see function `proximalFlat`)
- **`lambda1`** – (regularization parameter)
- **`lambda2`** – (optional, regularization parameter, 0 by default)
- **`lambda3`** – (optional, regularization parameter, 0 by default)
- **`verbose`** – (optional, verbosity level, false by default)
- **`pos`** – (optional, adds positivity constraints on the coefficients, false by default)
- **`transpose`** – (optional, transpose the matrix in the regularization function)
- **`size_group`** – (optional, for regularization functions assuming a group structure)
- **`groups`** – (int32, optional, for regularization functions assuming a group structure, see `proximalFlat`)
- **`numThreads`** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **`max_it`** – (optional, maximum number of iterations, 100 by default)
- **`it0`** – (optional, frequency for computing duality gap, every 10 iterations by default)
- **`tol`** – (optional, tolerance for stopping criterion, which is a relative duality gap if it is available, or a relative change of parameters).
- **`gamma`** – (optional, multiplier for increasing the parameter L in `fista`, 1.5 by default)
- **`L0`** – (optional, initial parameter L in `fista`, 0.1 by default, should be small enough)
- **`fixed_step`** – (deactive the line search for L in `fista` and use $L0$ instead)
- **`linesearch_mode`** – (line-search scheme when `ista=true`):
 - **`0`** – default, monotonic backtracking scheme
 - **`1`** – monotonic backtracking scheme, with restart at each iteration

- **2** – Barzilai-Borwein step sizes (similar to SparSA by Wright et al.)
- **3** – non-monotonic backtracking
- **compute_gram** – (optional, pre-compute $X^T X$, false by default).
- **intercept** – (optional, do not regularize last row of W , false by default).
- **ista** – (optional, use ista instead of fista, false by default).
- **subgrad** – (optional, if not ista, use subgradient descent instead of fista, false by default).
- **a** –
- **b** – (optional, if subgrad, the gradient step is $a/(t+b)$ also similar options as proximalFlat
the function also implements the ADMM algorithm via an option `admm=true`. It is not documented and you need to look at the source code to use it.
- **delta** – undocumented; modify at your own risks!
- **c** – undocumented; modify at your own risks!
- **max_iter_backtracking** – undocumented; modify at your own risks!
- **lin_admm** – undocumented; modify at your own risks!
- **admm** – undocumented; modify at your own risks!
- **resetflow** – undocumented; modify at your own risks!
- **clever** – undocumented; modify at your own risks!
- **log** – undocumented; modify at your own risks!
- **logName** – undocumented; modify at your own risks!
- **is_inner_weights** – undocumented; modify at your own risks!
- **inner_weights** – undocumented; modify at your own risks!
- **sqrt_step** – undocumented; modify at your own risks!

Returns

- **W**: double dense $p \times n$ matrix or $p \times N_n$ matrix (for multi-logistic loss)
- **optim**: optional, double dense $4 \times n$ matrix. first row: values of the objective functions. third row: values of the relative duality gap (if available) fourth row: number of iterations
- **optim_info**: vector of size 4, containing information of the optimization.
 $W = \text{spams.fistaFlat}(Y, X, W0, \text{return_optim_info} = \text{False}, \dots)$ $(W, \text{optim_info}) = \text{spams.fistaFlat}(Y, X, W0, \text{return_optim_info} = \text{True}, \dots)$

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: Valid values for the regularization parameter (`regul`) are: “l0”, “l1”, “l2”, “linf”, “l2-not-squared”, “elastic-net”, “fused-lasso”, “group-lasso-l2”, “group-lasso-linf”, “sparse-group-lasso-l2”, “sparse-group-lasso-linf”, “l1l2”, “l1linf”, “l1l2+l1”, “l1linf+l1”, “tree-l0”, “tree-l2”, “tree-linf”, “graph”, “graph-ridge”, “graph-l2”, “multi-task-tree”, “multi-task-graph”, “l1linf-row-column”, “trace-norm”, “trace-norm-vec”, “rank”, “rank-vec”, “none”

FISTATREE

```
spams.fistaTree(Y, X, W0, tree, return_optim_info=False, numThreads=-1, max_it=1000, L0=1.0,
               fixed_step=False, gamma=1.5, lambda1=1.0, delta=1.0, lambda2=0.0, lambda3=0.0,
               a=1.0, b=0.0, c=1.0, tol=1e-06, it0=100, max_iter_backtracking=1000, compute_gram=False,
               lin_admm=False, admm=False, intercept=False, resetflow=False, regul='', loss='',
               verbose=False, pos=False, clever=False, log=False, ista=False, subgrad=False,
               logName='', is_inner_weights=False, inner_weights=None, size_group=1, sqrt_step=True,
               transpose=False, linesearch_mode=0)
```

fistaTree solves sparse regularized problems.

X is a design matrix of size $m \times p$ $X=[x^1, \dots, x^n]^T$, where the x_i 's are the rows of X $Y=[y^1, \dots, y^n]$ is a matrix of size $m \times n$ It implements the algorithms FISTA, ISTA and subgradient descent for solving

$$\min_W \text{loss}(W) + \lambda \text{psi}(W)$$

The function `psi` are those used by `proximalTree` (see documentation) for the loss functions, see the documentation of `fistaFlat`

This function can also handle intercepts (last row of W is not regularized), and/or non-negativity constraints on W and sparse matrices X

Parameters

- **Y** – double dense $m \times n$ matrix
- **X** – double dense or sparse $m \times p$ matrix
- **W0** – double dense $p \times n$ matrix or $p \times N_n$ matrix (for multi-logistic loss) initial guess
- **tree** – named list (see documentation of `proximalTree`)
- **return_optim_info** – if true the function will return a tuple of matrices.
- **loss** – (choice of loss, see above)
- **regul** – (choice of regularization, see function `proximalFlat`)
- **lambda1** – (regularization parameter)
- **lambda2** – (optional, regularization parameter, 0 by default)
- **lambda3** – (optional, regularization parameter, 0 by default)
- **verbose** – (optional, verbosity level, false by default)
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)
- **transpose** – (optional, transpose the matrix in the regularization function)
- **size_group** – (optional, for regularization functions assuming a group structure)

- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **max_it** – (optional, maximum number of iterations, 100 by default)
- **it0** – (optional, frequency for computing duality gap, every 10 iterations by default)
- **tol** – (optional, tolerance for stopping criterion, which is a relative duality gap if it is available, or a relative change of parameters).
- **gamma** – (optional, multiplier for increasing the parameter L in fista, 1.5 by default)
- **L0** – (optional, initial parameter L in fista, 0.1 by default, should be small enough)
- **fixed_step** – (deactivate the line search for L in fista and use L0 instead)
- **compute_gram** – (optional, pre-compute $X^T X$, false by default).
- **intercept** – (optional, do not regularize last row of W, false by default).
- **ista** – (optional, use ista instead of fista, false by default).
- **subgrad** – (optional, if not ista, use subgradient descent instead of fista, false by default).
- **a** –
- **b** – (optional, if subgrad, the gradient step is $a/(t+b)$ also similar options as proximalTree
the function also implements the ADMM algorithm via an option `admm=true`. It is not documented and you need to look at the source code to use it.
- **delta** – undocumented; modify at your own risks!
- **c** – undocumented; modify at your own risks!
- **max_iter_backtracking** – undocumented; modify at your own risks!
- **lin_admm** – undocumented; modify at your own risks!
- **admm** – undocumented; modify at your own risks!
- **resetflow** – undocumented; modify at your own risks!
- **clever** – undocumented; modify at your own risks!
- **log** – undocumented; modify at your own risks!
- **logName** – undocumented; modify at your own risks!
- **is_inner_weights** – undocumented; modify at your own risks!
- **inner_weights** – undocumented; modify at your own risks!
- **sqrstep** – undocumented; modify at your own risks!

Returns

- **W**: double dense $p \times n$ matrix or $p \times N_n$ matrix (for multi-logistic loss)
- **optim**: optional, double dense $4 \times n$ matrix. first row: values of the objective functions. third row: values of the relative duality gap (if available) fourth row: number of iterations
- **optim_info**: vector of size 4, containing information of the optimization. `W = spams.fistaTree(Y,X,W0,tree,return_optim_info = False,...)` (`W,optim_info`) = `spams.fistaTree(Y,X,W0,tree,return_optim_info = True,...)`

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)

- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: Valid values for the regularization parameter (regul) are: "l0", "l1", "l2", "l1f", "l2-not-squared", "elastic-net", "fused-lasso", "group-lasso-l2", "group-lasso-l1f", "sparse-group-lasso-l2", "sparse-group-lasso-l1f", "l1l2", "l1l1f", "l1l2+l1", "l1l1f+l1", "tree-l0", "tree-l2", "tree-l1f", "graph", "graph-ridge", "graph-l2", "multi-task-tree", "multi-task-graph", "l1l1f-row-column", "trace-norm", "trace-norm-vec", "rank", "rank-vec", "none"

FISTAGRAPH

```
spams.fistaGraph(Y, X, W0, graph, return_optim_info=False, numThreads=-1, max_it=1000,
                 L0=1.0, fixed_step=False, gamma=1.5, lambda1=1.0, delta=1.0,
                 lambda2=0.0, lambda3=0.0, a=1.0, b=0.0, c=1.0, tol=1e-06, it0=100,
                 max_iter_backtracking=1000, compute_gram=False, lin_admm=False,
                 admm=False, intercept=False, resetflow=False, regul='', loss='', verbose=False,
                 pos=False, clever=False, log=False, ista=False, subgrad=False, logName='',
                 is_inner_weights=False, inner_weights=None, size_group=1, sqrt_step=True,
                 transpose=False, linesearch_mode=0)
```

fistaGraph solves sparse regularized problems.

X is a design matrix of size $m \times p$ $X=[x^1, \dots, x^n]'$, where the x_i 's are the rows of X $Y=[y^1, \dots, y^n]$ is a matrix of size $m \times n$ It implements the algorithms FISTA, ISTA and subgradient descent.

It implements the algorithms FISTA, ISTA and subgradient descent for solving

$$\min_W \text{loss}(W) + \lambda \text{psi}(W)$$

The function `psi` are those used by `proximalGraph` (see documentation) for the loss functions, see the documentation of `fistaFlat`

This function can also handle intercepts (last row of W is not regularized), and/or non-negativity constraints on W .

Parameters

- **Y** – double dense $m \times n$ matrix
- **X** – double dense or sparse $m \times p$ matrix
- **W0** – double dense $p \times n$ matrix or $p \times N_n$ matrix (for multi-logistic loss) initial guess
- **graph** – struct (see documentation of `proximalGraph`)
- **return_optim_info** – if true the function will return a tuple of matrices.
- **loss** – (choice of loss, see above)
- **regul** – (choice of regularization, see function `proximalFlat`)
- **lambda1** – (regularization parameter)
- **lambda2** – (optional, regularization parameter, 0 by default)
- **lambda3** – (optional, regularization parameter, 0 by default)
- **verbose** – (optional, verbosity level, false by default)
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)

- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **max_it** – (optional, maximum number of iterations, 100 by default)
- **it0** – (optional, frequency for computing duality gap, every 10 iterations by default)
- **tol** – (optional, tolerance for stopping criterion, which is a relative duality gap if it is available, or a relative change of parameters).
- **gamma** – (optional, multiplier for increasing the parameter L in fista, 1.5 by default)
- **L0** – (optional, initial parameter L in fista, 0.1 by default, should be small enough)
- **fixed_step** – (deactive the line search for L in fista and use L0 instead)
- **compute_gram** – (optional, pre-compute $X^T X$, false by default).
- **intercept** – (optional, do not regularize last row of W, false by default).
- **ista** – (optional, use ista instead of fista, false by default).
- **subgrad** – (optional, if not ista, use subgradient descent instead of fista, false by default).
- **a** –
- **b** – (optional, if subgrad, the gradient step is $a/(t+b)$ also similar options as proximalTree
the function also implements the ADMM algorithm via an option `admm=true`. It is not documented and you need to look at the source code to use it.
- **delta** – undocumented; modify at your own risks!
- **c** – undocumented; modify at your own risks!
- **max_iter_backtracking** – undocumented; modify at your own risks!
- **lin_admm** – undocumented; modify at your own risks!
- **admm** – undocumented; modify at your own risks!
- **resetflow** – undocumented; modify at your own risks!
- **clever** – undocumented; modify at your own risks!
- **log** – undocumented; modify at your own risks!
- **logName** – undocumented; modify at your own risks!
- **is_inner_weights** – undocumented; modify at your own risks!
- **inner_weights** – undocumented; modify at your own risks!
- **sqrstep** – undocumented; modify at your own risks!
- **size_group** – undocumented; modify at your own risks!
- **transpose** – undocumented; modify at your own risks!

Returns

- **W**: double dense $p \times n$ matrix or $p \times N_n$ matrix (for multi-logistic loss)
- **optim**: optional, double dense $4 \times n$ matrix. first row: values of the objective functions. third row: values of the relative duality gap (if available) fourth row: number of iterations
- **optim_info**: vector of size 4, containing information of the optimization. `W = spams.fistaGraph(Y,X,W0,graph,return_optim_info = False,...)` (`W,optim_info`) = `spams.fistaGraph(Y,X,W0,graph,return_optim_info = True,...)`

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: Valid values for the regularization parameter (regul) are: "l0", "l1", "l2", "l1f", "l2-not-squared", "elastic-net", "fused-lasso", "group-lasso-l2", "group-lasso-l1f", "sparse-group-lasso-l2", "sparse-group-lasso-l1f", "l1l2", "l1l1f", "l1l2+l1", "l1l1f+l1", "tree-l0", "tree-l2", "tree-l1f", "graph", "graph-ridge", "graph-l2", "multi-task-tree", "multi-task-graph", "l1l1f-row-column", "trace-norm", "trace-norm-vec", "rank", "rank-vec", "none"

PROXIMALFLAT

`spams.proximalFlat` (*U*, *return_val_loss=False*, *numThreads=-1*, *lambda1=1.0*, *lambda2=0.0*,
lambda3=0.0, *intercept=False*, *resetflow=False*, *regul=''*, *verbose=False*,
pos=False, *clever=True*, *size_group=1*, *groups=None*, *transpose=False*)

proximalFlat computes proximal operators. Depending on the value of *regul*, it computes

Given an input matrix $U=[u^1, \dots, u^n]$, it computes a matrix $V=[v^1, \dots, v^n]$ such that if one chooses a regularization functions on vectors, it computes for each column u of U , a column v of V solving if *regul*='l0'

$$\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \text{card}(v)$$

if *regul*='l1' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \|v\|_1$

if *regul*='l2' $\operatorname{argmin} 0.5\|u-v\|_2^2 + 0.5\lambda \|v\|_2^2$

if *regul*='elastic-net' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \|v\|_1 + \lambda_2 \|v\|_2^2$

if *regul*='fused-lasso'

$$\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \text{FL}(v) + \dots \dots \lambda_2 \|v\|_1 + \lambda_3 \|v\|_2^2$$

if *regul*='linf' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \|v\|_\infty$

if *regul*='l1-constraint' $\operatorname{argmin} 0.5\|u-v\|_2^2$ s.t. $\|v\|_1 \leq \lambda$

if *regul*='l2-not-squared' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \|v\|_2$

if *regul*='group-lasso-l2' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_g \|v_g\|_2$ where the groups are either defined by *groups* or by *size_group*,

if *regul*='group-lasso-linf' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_g \|v_g\|_\infty$

if *regul*='sparse-group-lasso-l2' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_g \|v_g\|_2 + \lambda_2 \|v\|_1$ where the groups are either defined by *groups* or by *size_group*,

if *regul*='sparse-group-lasso-linf' $\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_g \|v_g\|_\infty + \lambda_2 \|v\|_1$

if *regul*='trace-norm-vec'

$$\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \|\text{mat}(v)\|_*$$

where $\text{mat}(v)$ has *size_group* rows

if one chooses a regularization function on matrices if *regul*='l1l2', $V =$

$$\operatorname{argmin} 0.5\|U-V\|_F^2 + \lambda \|V\|_{\{1/2\}}$$

if *regul*='l1linf', $V = \operatorname{argmin} 0.5\|U-V\|_F^2 + \lambda \|V\|_{\{1/\infty\}}$

if *regul*='l1l2+l1', $V = \operatorname{argmin} 0.5\|U-V\|_F^2 + \lambda \|V\|_{\{1/2\}} + \lambda_2 \|V\|_{\{1/1\}}$

```
if regul='l1linf+l1', V= argmin 0.5||U-V||_F^2 + lambda1||V||_{1/inf} + lambda1_2||V||_{1/1}
if regul='l1linf+row-column', V= argmin 0.5||U-V||_F^2 + lambda1||V||_{1/inf} +
    lambda1_2||V'||_{1/inf}
if regul='trace-norm', V= argmin 0.5||U-V||_F^2 + lambda1||V||_*
if regul='rank', V= argmin 0.5||U-V||_F^2 + lambda1 rank(V)
if regul='none', V= argmin 0.5||U-V||_F^2
```

for all these regularizations, it is possible to enforce non-negativity constraints with the option `pos`, and to prevent the last row of `U` to be regularized, with the option `intercept`

Parameters

- **U** – double $m \times n$ matrix (input signals) m is the signal size
- **return_val_loss** – if true the function will return a tuple of matrices.
- **lambda1** – (regularization parameter)
- **regul** – (choice of regularization, see above)
- **lambda2** – (optional, regularization parameter)
- **lambda3** – (optional, regularization parameter)
- **verbose** – (optional, verbosity level, false by default)
- **intercept** – (optional, last row of `U` is not regularized, false by default)
- **transpose** – (optional, transpose the matrix in the regularization function)
- **size_group** – (optional, for regularization functions assuming a group structure). It is a scalar. When groups is not specified, it assumes that the groups are the sets of consecutive elements of size `size_group`
- **groups** – (int32, optional, for regularization functions assuming a group structure. It is an int32 vector of size m containing the group indices of the variables (first group is 1).
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **resetflow** – undocumented; modify at your own risks!
- **clever** – undocumented; modify at your own risks!

Returns

- **V**: double $m \times n$ matrix (output coefficients)
- **val_regularizer**: double $1 \times n$ vector (value of the regularization term at the optimum).
- **val_loss**: vector of size `U.shape[1]` `alpha = spams.proximalFlat(U,return_val_loss = False,...)` (`alpha,val_loss = spams.proximalFlat(U,return_val_loss = True,...)`)

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: Valid values for the regularization parameter (regul) are: “l0”, “l1”, “l2”, “l1f”, “l2-not-squared”, “elastic-net”, “fused-lasso”, “group-lasso-l2”, “group-lasso-l1f”, “sparse-group-lasso-l2”, “sparse-group-lasso-l1f”, “l1l2”, “l1l1f”, “l1l2+l1”, “l1l1f+l1”, “tree-l0”, “tree-l2”, “tree-l1f”, “graph”, “graph-ridge”, “graph-l2”, “multi-task-tree”, “multi-task-graph”, “l1l1f-row-column”, “trace-norm”, “trace-norm-vec”, “rank”, “rank-vec”, “none”

PROXIMALTREE

```
spams.proximalTree(U, tree, return_val_loss=False, numThreads=-1, lambda1=1.0, lambda2=0.0,
                    lambda3=0.0, intercept=False, resetflow=False, regul='', verbose=False,
                    pos=False, clever=True, size_group=1, transpose=False)
```

proximalTree computes a proximal operator. Depending on the value of regul, it computes

Given an input matrix $U=[u^1, \dots, u^n]$, and a tree-structured set of groups T , it returns a matrix $V=[v^1, \dots, v^n]$:

when the regularization function is for vectors, for every column u of U , it compute a column v of V solving if regul='tree-l0'

$$\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_{g \in T} \delta^g(v)$$

if regul='tree-l2'

$$\text{for all } i, v^i = \operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_{g \in T} \eta_g \|v\|_2$$

if regul='tree-linf'

$$\text{for all } i, v^i = \operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda \sum_{g \in T} \eta_g \|v\|_\infty$$

when the regularization function is for matrices: if regul='multi-task-tree'

$$V = \operatorname{argmin} 0.5\|U-V\|_F^2 + \lambda_1 \sum_{i=1}^n \sum_{g \in T} \eta_g \|v^i\|_\infty + \dots \\ \lambda_2 \sum_{g \in T} \eta_g \max_{j \in g} \|V_{j\cdot}\|_\infty$$

it can also be used with any non-tree-structured regularization addressed by proximalFlat

for all these regularizations, it is possible to enforce non-negativity constraints with the option pos, and to prevent the last row of U to be regularized, with the option intercept

Parameters

- **U** – double $m \times n$ matrix (input signals) m is the signal size
- **tree** – named list with four fields, eta_g, groups, own_variables and N_own_variables.

The tree structure requires a particular organization of groups and variables * Let us denote by $N = \text{length}(T)$, the number of groups. the groups should be ordered $T=\{g_1, g_2, \dots, g_N\}$ such that if g_i is included in g_j , then $j \leq i$. g_1 should be the group at the root of the tree and contains every variable. * Every group is a set of contiguous indices for instance $g_i=\{3,4,5\}$ or $g_i=\{4,5,6,7\}$ or $g_i=\{4\}$, but not $\{3,5\}$; * We define $\text{root}(g_i)$ as the indices of the variables that are in g_i , but not in its descendants. For instance for $T=\{g_1=\{1,2,3,4\}, g_2=\{2,3\}, g_3=\{4\}\}$, then, $\text{root}(g_1)=\{1\}$, $\text{root}(g_2)=\{2,3\}$, $\text{root}(g_3)=\{4\}$, We assume that for all i , $\text{root}(g_i)$ is a set of contiguous variables * We assume that the smallest of $\text{root}(g_i)$ is also the smallest index of g_i .

For instance, $T = \{ g_1 = \{1, 2, 3, 4\}, g_2 = \{2, 3\}, g_3 = \{4\} \}$, is a valid set of groups. but we can not have $T = \{ g_1 = \{1, 2, 3, 4\}, g_2 = \{1, 2\}, g_3 = \{3\} \}$, since $\text{root}(g_1) = \{4\}$ and 4 is not the smallest element in g_1 .

We do not lose generality with these assumptions since they can be fulfilled for any tree-structured set of groups after a permutation of variables and a correct ordering of the groups. see more examples in `test_ProximalTree.m` of valid tree-structured sets of groups.

The first field sets the weights for every group `tree['eta_g']` double N vector

The next field sets inclusion relations between groups (but not between groups and variables): `tree['groups']` sparse (double or boolean) $N \times N$ matrix the (i, j) entry is non-zero if and only if i is different than j and g_i is included in g_j . the first column corresponds to the group at the root of the tree.

The next field define the smallest index of each group g_i , which is also the smallest index of $\text{root}(g_i)$ `tree['own_variables']` int32 N vector

The next field define for each group g_i , the size of $\text{root}(g_i)$ `tree['N_own_variables']` int32 N vector

examples are given in `test_ProximalTree.m`

- **return_val_loss** – if true the function will return a tuple of matrices.
- **lambda1** – (regularization parameter)
- **regul** – (choice of regularization, see above)
- **lambda2** – (optional, regularization parameter)
- **lambda3** – (optional, regularization parameter)
- **verbose** – (optional, verbosity level, false by default)
- **intercept** – (optional, last row of U is not regularized, false by default)
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)
- **transpose** – (optional, transpose the matrix in the regularization function)
- **size_group** – (optional, for regularization functions assuming a group structure). It is a scalar. When groups is not specified, it assumes that the groups are the sets of consecutive elements of size `size_group`
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **resetflow** – undocumented; modify at your own risks!
- **clever** – undocumented; modify at your own risks!

Returns

- **V**: double $m \times n$ matrix (output coefficients)
- **val_regularizer**: double $1 \times n$ vector (value of the regularization term at the optimum).
- **val_loss**: vector of size `U.shape[1]` `alpha = spams.proximalTree(U, tree, return_val_loss = False, ...)` (`alpha, val_loss`) = `spams.proximalTree(U, tree, return_val_loss = True, ...)`

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: Valid values for the regularization parameter (regul) are: “l0”, “l1”, “l2”, “l1f”, “l2-not-squared”, “elastic-net”, “fused-lasso”, “group-lasso-l2”, “group-lasso-l1f”, “sparse-group-lasso-l2”, “sparse-group-lasso-l1f”, “l1l2”, “l1l1f”, “l1l2+l1”, “l1l1f+l1”, “tree-l0”, “tree-l2”, “tree-l1f”, “graph”, “graph-ridge”, “graph-l2”, “multi-task-tree”, “multi-task-graph”, “l1l1f-row-column”, “trace-norm”, “trace-norm-vec”, “rank”, “rank-vec”, “none”

PROXIMALGRAPH

`spams.proximalGraph` (*U*, *graph*, *return_val_loss=False*, *numThreads=-1*, *lambda1=1.0*, *lambda2=0.0*, *lambda3=0.0*, *intercept=False*, *resetflow=False*, *regul=''*, *verbose=False*, *pos=False*, *clever=True*, *eval=None*, *size_group=1*, *transpose=False*)

proximalGraph computes a proximal operator. Depending on the value of *regul*, it computes

Given an input matrix $U=[u^1, \dots, u^n]$, and a set of groups G , it computes a matrix $V=[v^1, \dots, v^n]$ such that

if *regul*='graph' for every column u of U , it computes a column v of V solving

$$\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda_1 \sum_{g \in G} \eta_g \|v\|_{\infty}$$

if *regul*='graph+ridge' for every column u of U , it computes a column v of V solving

$$\operatorname{argmin} 0.5\|u-v\|_2^2 + \lambda_1 \sum_{g \in G} \eta_g \|v\|_{\infty} + \lambda_2 \|v\|_2^2$$

if *regul*='multi-task-graph'

$$V = \operatorname{argmin} 0.5\|U-V\|_F^2 + \lambda_1 \sum_{i=1}^n \sum_{g \in G} \eta_g \|v^i\|_{\infty} + \dots \\ \lambda_2 \sum_{g \in G} \sum_{j \in g} \eta_g \max_{j \in g} \|V_j\|_{\infty}$$

it can also be used with any regularization addressed by `proximalFlat`

for all these regularizations, it is possible to enforce non-negativity constraints with the option *pos*, and to prevent the last row of U to be regularized, with the option *intercept*

Parameters

- **U** – double $p \times n$ matrix (input signals) m is the signal size
- **graph** – struct with three fields, *eta_g*, *groups*, and *groups_var*

The first field sets the weights for every group *graph.eta_g* double N vector

The next field sets inclusion relations between groups (but not between groups and variables): *graph.groups* sparse (double or boolean) $N \times N$ matrix the (i,j) entry is non-zero if and only if i is different than j and g_i is included in g_j .

The next field sets inclusion relations between groups and variables *graph.groups_var* sparse (double or boolean) $p \times N$ matrix the (i,j) entry is non-zero if and only if the variable i is included in g_j , but not in any children of g_j .

examples are given in `test_ProximalGraph.m`

- **return_val_loss** – if true the function will return a tuple of matrices.
- **lambda1** – (regularization parameter)
- **regul** – (choice of regularization, see above)

- **lambda2** – (optional, regularization parameter)
- **lambda3** – (optional, regularization parameter)
- **verbose** – (optional, verbosity level, false by default)
- **intercept** – (optional, last row of U is not regularized, false by default)
- **pos** – (optional, adds positivity constraints on the coefficients, false by default)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **resetflow** – undocumented; modify at your own risks!
- **clever** – undocumented; modify at your own risks!
- **size_group** – undocumented; modify at your own risks!
- **transpose** – undocumented; modify at your own risks!

Returns

- **V**: double p x n matrix (output coefficients)
- **val_regularizer**: double 1 x n vector (value of the regularization term at the optimum).
- **val_loss**: vector of size U.shape[1] `alpha = spams.proximalGraph(U,graph,return_val_loss = False,...)` (`alpha,val_loss = spams.proximalGraph(U,graph,return_val_loss = True,...)`)

Authors

- Julien MAIRAL, 2010 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: Valid values for the regularization parameter (regul) are: “l0”, “l1”, “l2”, “l1f”, “l2-not-squared”, “elastic-net”, “fused-lasso”, “group-lasso-l2”, “group-lasso-l1f”, “sparse-group-lasso-l2”, “sparse-group-lasso-l1f”, “l1l2”, “l1l1f”, “l1l2+l1”, “l1l1f+l1”, “tree-l0”, “tree-l2”, “tree-l1f”, “graph”, “graph-ridge”, “graph-l2”, “multi-task-tree”, “multi-task-graph”, “l1l1f-row-column”, “trace-norm”, “trace-norm-vec”, “rank”, “rank-vec”, “none”

TRAINDL

`spams.trainDL(X, return_model=False, model=None, D=None, numThreads=-1, batchsize=-1, K=-1, lambda1=None, lambda2=1e-09, iter=-1, t0=1e-05, mode=2, posAlpha=False, posD=False, expand=False, modeD=0, whiten=False, clean=True, verbose=True, gamma1=0.0, gamma2=0.0, rho=1.0, iter_updateD=None, stochastic_deprecated=False, modeParam=0, batch=False, log_deprecated=False, logName='')`

`trainDL` is an efficient implementation of the dictionary learning technique presented in

“Online Learning for Matrix Factorization and Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro arXiv:0908.0050

“Online Dictionary Learning for Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro ICML 2009.

Note that if you use `mode=1` or `2`, if the training set has a reasonable size and you have enough memory on your computer, you should use `trainDL_Memory` instead.

It addresses the dictionary learning problems

1. if `mode=0`

$$\min_{\{D \in C\}} (1/n) \sum_{i=1}^n (1/2) \|x_i - D\alpha_i\|_2^2 \text{ s.t. } \dots$$
$$\|\alpha_i\|_1 \leq \lambda$$

2. if `mode=1`

$$\min_{\{D \in C\}} (1/n) \sum_{i=1}^n \|\alpha_i\|_1 \text{ s.t. } \dots$$
$$\|x_i - D\alpha_i\|_2^2 \leq \lambda$$

3. if `mode=2`

$$\min_{\{D \in C\}} (1/n) \sum_{i=1}^n (1/2) \|x_i - D\alpha_i\|_2^2 + \dots$$
$$\lambda \|\alpha_i\|_1 + \lambda_2 \|\alpha_i\|_2^2$$

4. if `mode=3`, the sparse coding is done with OMP

$$\min_{\{D \in C\}} (1/n) \sum_{i=1}^n (1/2) \|x_i - D\alpha_i\|_2^2 \text{ s.t. } \dots$$
$$\|\alpha_i\|_0 \leq \lambda$$

5. if `mode=4`, the sparse coding is done with OMP

$$\min_{\{D \in C\}} (1/n) \sum_{i=1}^n \|\alpha_i\|_0 \text{ s.t. } \dots$$
$$\|x_i - D\alpha_i\|_2^2 \leq \lambda$$

6. if mode=5, the sparse coding is done with OMP

$$\min_{\{D \in C\}} (1/n) \sum_{i=1}^n 0.5 \|x_i - D\alpha_i\|_2^2 + \lambda \|\alpha_i\|_1$$

C is a convex set verifying

1. if modeD=0 $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_{j\cdot}\|_2^2 \leq 1 \}$
2. if modeD=1 $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_{j\cdot}\|_2^2 + \dots + \gamma_1 \|d_{j1}\|_1 \leq 1 \}$
3. if modeD=2 $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_{j\cdot}\|_2^2 + \dots + \gamma_1 \|d_{j1}\|_1 + \gamma_2 \text{FL}(d_{j\cdot}) \leq 1 \}$
4. if modeD=3 $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, (1-\gamma_1)\|d_{j\cdot}\|_2^2 + \dots + \gamma_1 \|d_{j1}\|_1 \leq 1 \}$

Potentially, n can be very large with this algorithm.

Parameters

- **X** – double m x n matrix (input signals) m is the signal size n is the number of signals to decompose
- **return_model** – if true the function will return the model as a named list ('A' = A, 'B' = B, 'iter' = n)
- **model** – None or model (as A,B,iter) to use as initialisation
- **D** – (optional) double m x p matrix (dictionary) p is the number of elements in the dictionary When D is not provided, the dictionary is initialized with random elements from the training set.
- **K** – (size of the dictionary, optional is D is provided)
- **lambda1** – (parameter)
- **lambda2** – (optional, by default 0)
- **iter** – (number of iterations). If a negative number is provided it will perform the computation during the corresponding number of seconds. For instance iter=-5 learns the dictionary during 5 seconds.
- **mode** – (optional, see above, by default 2)
- **posAlpha** – (optional, adds positivity constraints on the coefficients, false by default, not compatible with mode =3,4)
- **modeD** – (optional, see above, by default 0)
- **posD** – (optional, adds positivity constraints on the dictionary, false by default, not compatible with modeD=2)
- **gamma1** – (optional parameter for modeD >= 1)
- **gamma2** – (optional parameter for modeD = 2)
- **batchsize** – (optional, size of the minibatch, by default 512)
- **iter_updatedD** – (optional, number of BCD iterations for the dictionary update step, by default 1)

- **modeParam** – (optimization mode). 1) if modeParam=0, the optimization uses the parameter free strategy of the ICML paper 2) if modeParam=1, the optimization uses the parameters rho as in arXiv:0908.0050 3) if modeParam=2, the optimization uses exponential decay weights with updates of the form $A_{\{t\}} \leftarrow \rho A_{\{t-1\}} + \alpha_t \alpha_t^T$
- **rho** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **t0** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **clean** – (optional, true by default. prunes automatically the dictionary from unused elements).
- **verbose** – (optional, true by default, increase verbosity)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **expand** – undocumented; modify at your own risks!
- **whiten** – undocumented; modify at your own risks!
- **stochastic_deprecated** – undocumented; modify at your own risks!
- **batch** – undocumented; modify at your own risks!
- **log_deprecated** – undocumented; modify at your own risks!
- **logName** – undocumented; modify at your own risks!

Returns

- **D**: double m x p matrix (dictionary)
- **model**: the model as A B iter D = spams.trainDL(X,return_model = False,...) (D,model) = spams.trainDL(X,return_model = True,...)

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting

STRUCTTRAINDL

```
spams.structTrainDL(X, return_model=False, model=None, D=None, graph=None, tree=None,
                    numThreads=-1, tol=1e-06, fixed_step=True, ista=False, batchsize=-1,
                    K=-1, lambda1=None, lambda2=1e-09, lambda3=0.0, iter=-1, t0=1e-05,
                    regul='none', posAlpha=False, posD=False, expand=False, modeD=0,
                    whiten=False, clean=True, verbose=True, gamma1=0.0, gamma2=0.0, rho=1.0,
                    iter_updateD=None, stochastic_deprecated=False, modeParam=0, batch=False,
                    log_deprecated=False, logName='')
```

structTrainDL is an efficient implementation of the dictionary learning technique presented in

“Online Learning for Matrix Factorization and Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro arXiv:0908.0050

“Online Dictionary Learning for Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro ICML 2009.

It addresses the dictionary learning problems $\min_{D \in C} (1/n) \sum_{i=1}^n 0.5 \|x_i - D\alpha_i\|_2^2 + \lambda \psi(\alpha)$ where the regularization function ψ depends on `regul` (see `proximalFlat` for the description of ψ ,

and `regul` below for allowed values of `regul`)

C is a convex set verifying

1. if `modeD=0` $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_j\|_2^2 \leq 1 \}$
2. if `modeD=1` $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_j\|_2^2 + \dots$
 $\gamma_1 \|d_j\|_1 \leq 1 \}$
3. if `modeD=2` $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_j\|_2^2 + \dots$
 $\gamma_1 \|d_j\|_1 + \gamma_2 \text{FL}(d_j) \leq 1 \}$
4. if `modeD=3` $C = \{ D \in \mathbb{R}^{m \times p} \text{ s.t. } \forall j, (1-\gamma_1)\|d_j\|_2^2 + \dots$
 $\gamma_1 \|d_j\|_1 \leq 1 \}$

Potentially, n can be very large with this algorithm.

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **return_model** – if true the function will return the model as a named list (`'A'` = A , `'B'` = B , `'iter'` = n)
- **model** – None or model (as A, B, iter) to use as initialisation

- **D** – (optional) double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary. When **D** is not provided, the dictionary is initialized with random elements from the training set.
- **K** – (size of the dictionary, optional if **D** is provided)
- **lambda1** – (parameter)
- **lambda2** – (optional, by default 0)
- **lambda3** – (optional, regularization parameter, 0 by default)
- **iter** – (number of iterations). If a negative number is provided it will perform the computation during the corresponding number of seconds. For instance `iter=-5` learns the dictionary during 5 seconds.
- **regul** – choice of regularization : one of 'l0' 'l1' 'l2' 'linf' 'none' 'elastic-net' 'fused-lasso' 'graph' 'graph-ridge' 'graph-l2' 'tree-l0' 'tree-l2' 'tree-linf'
- **tree** – struct (see documentation of `proximalTree`); needed for regul of graph kind.
- **graph** – struct (see documentation of `proximalGraph`); needed for regul of tree kind.
- **posAlpha** – (optional, adds positivity constraints on the coefficients, false by default.
- **modeD** – (optional, see above, by default 0)
- **posD** – (optional, adds positivity constraints on the dictionary, false by default, not compatible with `modeD=2`)
- **gamma1** – (optional parameter for `modeD >= 1`)
- **gamma2** – (optional parameter for `modeD = 2`)
- **batchsize** – (optional, size of the minibatch, by default 512)
- **iter_updatedD** – (optional, number of BCD iterations for the dictionary update step, by default 1)
- **modeParam** – (optimization mode). 1) if `modeParam=0`, the optimization uses the parameter free strategy of the ICML paper 2) if `modeParam=1`, the optimization uses the parameters ρ as in arXiv:0908.0050 3) if `modeParam=2`, the optimization uses exponential decay weights with updates of the form $A_{\{t\}} \leftarrow \rho A_{\{t-1\}} + \alpha_t A_{\{t-1\}}^T$
- **ista** – (optional, use ista instead of fista, false by default).
- **tol** – (optional, tolerance for stopping criterion, which is a relative duality gap)
- **fixed_step** – (deactivate the line search for L in fista and use K instead)
- **rho** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **t0** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **clean** – (optional, true by default. prunes automatically the dictionary from unused elements).
- **verbose** – (optional, true by default, increase verbosity)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **expand** – undocumented; modify at your own risks!
- **whiten** – undocumented; modify at your own risks!
- **stochastic_deprecated** – undocumented; modify at your own risks!

- **batch** – undocumented; modify at your own risks!
- **log_deprecated** – undocumented; modify at your own risks!
- **logName** – undocumented; modify at your own risks!

Returns

- **D**: double $m \times p$ matrix (dictionary)
- **model**: the model as `A B iter D = spams.structTrainDL(X,return_model = False,...)`
(`D,model`) = `spams.structTrainDL(X,return_model = True,...)`

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting

TRAINDL_MEMORY

```
spams.trainDL_Memory(X, D=None, numThreads=-1, batchsize=-1, K=-1, lambda1=None, iter=-1,
                      t0=1e-05, mode=2, posD=False, expand=False, modeD=0, whiten=False,
                      clean=True, gamma1=0.0, gamma2=0.0, rho=1.0, iter_updateD=1, stochastic_deprecated=False,
                      modeParam=0, batch=False, log_deprecated=False, logName='')
```

trainDL_Memory is an efficient but memory consuming variant of the dictionary learning technique presented in

“Online Learning for Matrix Factorization and Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro arXiv:0908.0050

“Online Dictionary Learning for Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro ICML 2009.

Contrary to the approaches above, the algorithm here does require to store all the coefficients from all the training signals. For this reason this variant can not be used with large training sets, but is more efficient than the regular online approach for training sets of reasonable size.

It addresses the dictionary learning problems

1. if mode=1

$\min_{\{D \text{ in } C\}} (1/n) \sum_{i=1}^n \|\alpha_{i1}\|_1 \text{ s.t. } \dots$

$$\|x_i - D\alpha_{i1}\|_2^2 \leq \lambda$$

2. if mode=2

$\min_{\{D \text{ in } C\}} (1/n) \sum_{i=1}^n (1/2) \|x_i - D\alpha_{i1}\|_2^2 + \dots \lambda \|\alpha_{i1}\|_1$

C is a convex set verifying

1. if modeD=0 $C = \{ D \text{ in } \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_j\|_2^2 \leq 1 \}$

1. if modeD=1 $C = \{ D \text{ in } \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_j\|_2^2 + \dots \gamma_1 \|d_j\|_1 \leq 1 \}$

1. if modeD=2 $C = \{ D \text{ in } \mathbb{R}^{m \times p} \text{ s.t. } \forall j, \|d_j\|_2^2 + \dots \gamma_1 \|d_j\|_1 + \gamma_2 \text{FL}(d_j) \leq 1 \}$

Potentially, n can be very large with this algorithm.

Parameters

- **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose
- **D** – (optional) double $m \times p$ matrix (dictionary) p is the number of elements in the dictionary When D is not provided, the dictionary is initialized with random elements from the training set.
- **K** – (size of the dictionary, optional is D is provided)
- **lambda1** – (parameter)
- **iter** – (number of iterations). If a negative number is provided it will perform the computation during the corresponding number of seconds. For instance `iter=-5` learns the dictionary during 5 seconds.
- **mode** – (optional, see above, by default 2)
- **modeD** – (optional, see above, by default 0)
- **posD** – (optional, adds positivity constraints on the dictionary, false by default, not compatible with `modeD=2`)
- **gamma1** – (optional parameter for `modeD >= 1`)
- **gamma2** – (optional parameter for `modeD = 2`)
- **batchsize** – (optional, size of the minibatch, by default 512)
- **iter_updatedD** – (optional, number of BCD iterations for the dictionary update step, by default 1)
- **modeParam** – (optimization mode). 1) if `modeParam=0`, the optimization uses the parameter free strategy of the ICML paper 2) if `modeParam=1`, the optimization uses the parameters ρ as in arXiv:0908.0050 3) if `modeParam=2`, the optimization uses exponential decay weights with updates of the form $A_{\{t\}} \leftarrow \rho A_{\{t-1\}} + \alpha_t \alpha_t^T$
- **rho** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **t0** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **clean** – (optional, true by default. prunes automatically the dictionary from unused elements).
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **expand** – undocumented; modify at your own risks!
- **whiten** – undocumented; modify at your own risks!
- **stochastic_deprecated** – undocumented; modify at your own risks!
- **batch** – undocumented; modify at your own risks!
- **log_deprecated** – undocumented; modify at your own risks!
- **logName** – undocumented; modify at your own risks!

Returns

- **D**: double $m \times p$ matrix (dictionary)
- **model**: the model as $A B \text{ iter } D = \text{spams.trainDL_Memory}(X, \dots)$

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)

- Jean-Paul CHIEZE 2011-2012 (python interface)

Note: this function admits a few experimental usages, which have not been extensively tested: - single precision setting (even though the output alpha is double precision)

NMF

`spams.nmf` (*X*, *return_lasso=False*, *model=None*, *numThreads=-1*, *batchsize=-1*, *K=-1*, *iter=-1*, *t0=1e-05*,
clean=True, *rho=1.0*, *modeParam=0*, *batch=False*)

`trainDL` is an efficient implementation of the non-negative matrix factorization technique presented in

“Online Learning for Matrix Factorization and Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro arXiv:0908.0050

“Online Dictionary Learning for Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro ICML 2009.

Potentially, *n* can be very large with this algorithm.

Parameters

- **X** – double *m* x *n* matrix (input signals) *m* is the signal size *n* is the number of signals to decompose
- **return_lasso** – if true the function will return a tuple of matrices.
- **K** – (number of required factors)
- **iter** – (number of iterations). If a negative number is provided it will perform the computation during the corresponding number of seconds. For instance *iter=-5* learns the dictionary during 5 seconds.
- **batchsize** – (optional, size of the minibatch, by default 512)
- **modeParam** – (optimization mode). 1) if *modeParam=0*, the optimization uses the parameter free strategy of the ICML paper 2) if *modeParam=1*, the optimization uses the parameters *rho* as in arXiv:0908.0050 3) if *modeParam=2*, the optimization uses exponential decay weights with updates of the form $A_{\{t\}} \leftarrow \rho A_{\{t-1\}} + \alpha_t \alpha_t^T$
- **rho** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **t0** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **clean** – (optional, true by default. prunes automatically the dictionary from unused elements).
- **batch** – (optional, false by default, use batch learning instead of online learning)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **model** – struct (optional) learned model for “retraining” the data.

Returns

- **U**: double *m* x *p* matrix

- **V**: double $p \times n$ matrix (optional)
- **model**: struct (optional) learned model to be used for “retraining” the data. $U = \text{spams.nmf}(X, \text{return_lasso} = \text{False}, \dots)$ $(U, V) = \text{spams.nmf}(X, \text{return_lasso} = \text{True}, \dots)$

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

NNSC

`spams.nnsc` (*X*, *return_lasso=False*, *model=None*, *lambda1=None*, *numThreads=-1*, *batchsize=-1*, *K=-1*,
iter=-1, *t0=1e-05*, *clean=True*, *rho=1.0*, *modeParam=0*, *batch=False*)

`trainDL` is an efficient implementation of the non-negative sparse coding technique presented in

“Online Learning for Matrix Factorization and Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro arXiv:0908.0050

“Online Dictionary Learning for Sparse Coding” by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro ICML 2009.

Potentially, *n* can be very large with this algorithm.

Parameters

- **X** – double *m* x *n* matrix (input signals) *m* is the signal size *n* is the number of signals to decompose
- **return_lasso** – if true the function will return a tuple of matrices.
- **K** – (number of required factors)
- **lambda1** – (parameter)
- **iter** – (number of iterations). If a negative number is provided it will perform the computation during the corresponding number of seconds. For instance *iter=-5* learns the dictionary during 5 seconds.
- **batchsize** – (optional, size of the minibatch, by default 512)
- **modeParam** – (optimization mode). 1) if *modeParam=0*, the optimization uses the parameter free strategy of the ICML paper 2) if *modeParam=1*, the optimization uses the parameters *rho* as in arXiv:0908.0050 3) if *modeParam=2*, the optimization uses exponential decay weights with updates of the form $A_{\{t\}} \leftarrow \rho A_{\{t-1\}} + \alpha_t A_{\{t\}}^T$
- **rho** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **t0** – (optional) tuning parameter (see paper arXiv:0908.0050)
- **clean** – (optional, true by default. prunes automatically the dictionary from unused elements).
- **batch** – (optional, false by default, use batch learning instead of online learning)
- **numThreads** – (optional, number of threads for exploiting multi-core / multi-cpus. By default, it takes the value -1, which automatically selects all the available CPUs/cores).
- **model** – struct (optional) learned model for “retraining” the data.

Returns

- **U**: double $m \times p$ matrix
- **V**: double $p \times n$ matrix (optional)
- **model**: struct (optional) learned model to be used for “retraining” the data. `U = spams.nnsc(X,return_lasso = False,...)` `(U,V) = spams.nnsc(X,return_lasso = True,...)`

Authors

- Julien MAIRAL, 2009 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

ARCHETYPAL ANALYSIS

`spams.archetypalAnalysis` (X , $p=10$, $Z0=None$, $returnAB=False$, $robust=False$, $epsilon=0.001$,
 $computeXtX=False$, $stepsFISTA=3$, $stepsAS=50$, $randominit=False$,
 $numThreads=-1$)

documentation to appear soon

Parameters X – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose

Returns

- Z : double %

Authors

- Yuansi Chen and Julien MAIRAL, 2014 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

DECOMPSIMPLEX

`spams.decompSimplex(X, Z, computeXtX=False, numThreads=-1)`
documentation to appear soon

Parameters **X** – double $m \times n$ matrix (input signals) m is the signal size n is the number of signals to decompose

Returns

- **Z**: double %

Authors

- Yuansi Chen and Julien MAIRAL, 2014 (spams, matlab interface and documentation)
- Jean-Paul CHIEZE 2011-2012 (python interface)

SIMPLEGROUPTREE

`spams.simpleGroupTree` (*degrees*)

makes a structure representing a tree given the degree of each level.

Parameters `degrees` – int vector; `degrees(i)` is the number of children of each node at level `i`

Returns

- **group_struct**: list, one element for each node an element is itself a 4 elements list : `nodeid` (`int >= 0`), `weight` (`double`), array of vars attached to the node (here equal to `[nodeid]`), array of children (`nodeid`'s)

Authors

- Jean-Paul CHIEZE, 2012

READGROUPSTRUCT

`spams.readGroupStruct` (*file*)

reads a text file describing “simply” the structure of groups of variables needed by `proximalGraph`, `proximalTree`, `fistaGraph`, `fistaTree` and `structTrainDL` and builds the corresponding group structure.

weight is a float variables-list : a space separated list of integers, maybe empty,
but '[' and ']' must be present. Numbers in the range (0 - N_v-1)

children-list [a space separated list of node-id's] If the list is empty, '->' may be omitted.

The data must obey some rules []

- A group contains the variables of the corresponding node and of the whole subtree.
- Variables attached to a node are those that are not in the subtree.
- **If the data destination is a Graph, there may be several independant trees,** and a variable may appear in several trees.

If the destination is a Tree, there must be only one tree, the root node must have `id == 0` and each variable must appear only once.

Parameters `file` – the file name

Returns

- **groups:** list, one element for each node an element is itself a 4 elements list: `nodeid` (int ≥ 0), `weight` (double), array of vars of the node, array of children (nodeid's)

Authors

- Jean-Paul CHIEZE, 2012

GROUPSTRUCTOFSTRING

`spams.groupStructOfString(s)`

decode a multi-line string describing “simply” the structure of groups of variables needed by proximal-Graph, proximalTree, fistaGraph, fistaTree and structTrainDL and builds the corresponding group structure.

Each line describes a group of variables as a node of a tree. It has up to 4 fields separated by spaces:

node-id node-weight [variables-list] -> children-list

Let's define N_g = number of groups, and N_v = number of variables. node-id must be in the range (0 - N_g-1), and there must be N_g nodes weight is a float variables-list : a space separated list of integers, maybe empty,

but '[' and ']' must be present. Numbers in the range (0 - N_v-1)

children-list [a space separated list of node-id's] If the list is empty, '->' may be omitted.

The data must obey some rules []

- A group contains the variables of the corresponding node and of the whole subtree.
- Variables attached to a node are those that are not in the subtree.
- If the data destination is a Graph, there may be several independant trees, and a variable may appear in several trees.

If the destination is a Tree, there must be only one tree, the root node must have `id == 0` and each variable must appear only once.

Parameters `s` – the multi-lines string

Returns

- **groups**: list, one element for each node an element is itself a 4 elements list: nodeid (int ≥ 0), weight (double), array of vars of the node, array of children (nodeid's)

Authors

- Jean-Paul CHIEZE, 2012

GRAPHOFGROUPSTRUCT

`spams.graphOfGroupStruct` (*gstruct*)

converts a group structure into the graph structure used by `proximalGraph`, `fistaGraph` or `structTrainDL`

Parameters **gstruct** – the structure of groups as a list, one element per node an element is itself a 4 elements list: nodeid (≥ 0), weight (double), array of vars associated to the node, array of children (nodeis's)

Returns

- **graph**: struct (see documentation of `proximalGraph`)

Authors

- Jean-Paul CHIEZE, 2012

TREEOFGROUPSTRUCT

`spams.treeOfGroupStruct(gstruct)`

converts a group structure into the tree structure used by `proximalTree`, `fistaTree` or `structTrainDL`

Parameters `gstruct` – the structure of groups as a list, one element per node an element is itself a 4 elements list: `nodeid` (≥ 0), `weight` (double), array of vars associated to the node, array of children (`nodeis's`)

Returns

- **permutations**: permutation vector that must be applied to the result of the program using the tree. Empty if no permutation is needed.
- **tree**: named list (see documentation of `proximalTree`)
- **nbvars**: number of variables in the tree (`permutations,tree,nbvars`) = `spams.treeOfGroupStruct(gstruct)`

Authors

- Jean-Paul CHIEZE, 2012

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

A

archetypalAnalysis() (in module spams), 79

B

bayer() (in module spams), 15

C

calcAAt() (in module spams), 5
 calcXAt() (in module spams), 7
 calcXtY() (in module spams), 13
 calcXY() (in module spams), 9
 calcXYt() (in module spams), 11
 cd() (in module spams), 35
 conjGrad() (in module spams), 17

D

decompSimplex() (in module spams), 81

F

fistaFlat() (in module spams), 41
 fistaGraph() (in module spams), 49
 fistaTree() (in module spams), 45

G

graphOfGroupStruct() (in module spams), 89
 groupStructOfString() (in module spams), 87

I

invSym() (in module spams), 19

L

l1L2BCD() (in module spams), 39
 lasso() (in module spams), 25
 lassoMask() (in module spams), 27
 lassoWeighted() (in module spams), 29

N

nmf() (in module spams), 75
 nnsc() (in module spams), 77
 normalize() (in module spams), 21

O

omp() (in module spams), 31
 ompMask() (in module spams), 33

P

proximalFlat() (in module spams), 53
 proximalGraph() (in module spams), 61
 proximalTree() (in module spams), 57

R

readGroupStruct() (in module spams), 85

S

simpleGroupTree() (in module spams), 83
 somp() (in module spams), 37
 sort() (in module spams), 3
 sparseProject() (in module spams), 23
 structTrainDL() (in module spams), 67

T

trainDL() (in module spams), 63
 trainDL_Memory() (in module spams), 71
 treeOfGroupStruct() (in module spams), 91