pysmme

Release 1.0

Adam Lund

Mar 10, 2022

CONTENTS:

CHAPTER

ONE

PYSMME

1.1 pysmme package

1.1.1 pysmme.tools module

This module contains functionality for i) solving (fitting, calibrating) the soft maximin problem and ii) predicting from this fitted solution (model).

`pysmme.tools.`**`predict`**(*fit*, *x*)

Make Prediction From an smme Object.

Parameters

fit [smme_dict] The output from a `pysmme.tools.softmaximin` call

x [list, matrix or string] an object that should be like the input to the `pysmme.tools.softmaximin` call that produced the object `fit`. For general models a matrix with column dimension equal to that of the original input. For array models with custom design a list and with wavelet design the name of the wavelet used.

Returns

list A list of length `len(zeta)`. If `x` is an $k \times p$ matrix `x` each list item is an $k \times m_\zeta$ matrix containing the linear predictors computed for each model. If `x` is a string or a list of matrices each of size $k_i \times p_i$, each list item is an array of size $k_1 \times \cdots \times k_d \times m_\zeta$, $d \in \{1, 2, 3\}$, with the linear predictors computed for each model.

Notes

Given input `x` data this function computes the linear predictors using the fitted model coefficients supplied in the `object` which should be produced by `softmaximin`. If `object` is the result of fitting general type model `x` should be a $k \times p$ matrix (p is the number of model coefficients and k is the number of new data points). If `object` is the result of fitting a model with tensor design x` should be a list containing $k_i \times p_i, i = 1, 2, 3$ matrices (k_i is the number of new marginal data points in the i.

Examples

#array data ##size of example

```
>>> G = 3;
>>> n = np.array([65, 26, 13])
>>> p = np.array([13, 5, 4])
```

##marginal design matrices (Kronecker components)

```
>>> x = [None] * 3
>>> for i in range(len(x)):
>>> x[i] = np.random.normal(0, 1, (n[i], p[i]))
```

##common features and effects

```
>>> common_features = np.random.binomial(1, 0.1, np.prod(p)) #sparsity of common␣
↪effects
>>> common_effects = np.random.normal(size = np.prod(p)) * common_features
```

##group response

```
>>> y = np.zeros((n[0], n[1], n[2], G))
>>> for g in range(G):
>>>     bg = np.random.normal(0, 0.1, np.prod(p)) * (1 - common_features) + common_
↪effects
>>>     mu = RH(x[2], RH(x[1], RH(x[0], np.reshape(bg, (p[0], p[1], p[2]), "F") )))
>>>     y[:, :, :, g] = np.random.normal(0, 1, (n)) + mu
```

##fit model for range of lambda and zeta

```
>>> zeta = np.array([0.1, 1, 10, 100])
>>> fit = softmaximin(y, x, zeta = zeta, penalty = "lasso", alg = "npg")
>>> yhat = predict(fit, x)
```

#Array data and wavelets ##size of example

```
>>> G = 5;
>>> p = n = np.array([2**2, 2**3, 2**4])
```

##wavelet design

```
>>> x = "la8"
```

##common features and effects

```
>>> common_features = np.random.binomial(1, 0.1, np.prod(p)) #sparsity of common␣
↪effects
>>> common_effects = np.random.normal(size = np.prod(p)) * common_features
```

##group response

```
>>> y = np.zeros((n[0], n[1], n[2], G))
>>> for g in range(G):
>>>     bg = np.random.normal(0, 0.1, np.prod(p)) * (1 - common_features) + common_
↪effects
```

(continues on next page)

(continued from previous page)

```
>>>     mu = iwt(np.reshape(bg, (p[0], p[1], p[2]), "F"))
>>>     y[:, :, :, g] = np.random.normal(0, 1, (n)) + mu
```

##fit model for range of lambda and zeta

```
>>> zeta = np.array([0.1, 1, 10, 100])
>>> fit = softmaximin(y, x, zeta = zeta, penalty = "lasso", alg = "npg")
>>> modelno = 10
>>> zetano = 2
>>> yhat = predict(fit, x)
>>> yhat[zetano][:,:,:, modelno]
```

#Non-array data ##size of example

```
>>> G = 10
>>> n = np.random.choice(np.arange(100,500,1), G) #sample(100:500, G);
>>> p = 60
>>> x = [None] * G
```

##group design matrices

```
>>> for i in range(len(x)):
>>> x[i] = np.random.normal(0, 1, (n[i], p))
```

##common features and effects

```
>>> common_features = np.random.binomial(1, 0.1, np.prod(p)) #sparsity of common␣
↪effects
>>> common_effects = np.random.normal(size = np.prod(p)) * common_features
```

##group response

```
>>> y = [None] * G
>>> for g in range(G):
>>>     bg = np.random.normal(0, 0.5, np.prod(p)) * (1 - common_features) + common_
↪effects
>>>     mu = np.matmul(x[g], bg)
>>>     y[g] = np.random.normal(0, 1, n[g]) + mu
```

##fit model for range of lamb and zeta

```
>>> fit = softmaximin(y, x, zeta = zeta, penalty = "lasso", alg = "npg")
>>> yhat = predict(fit, x)
```

`pysmme.tools.`**`softmaximin`**(*y*, *x*, *zeta*, *penalty*, *alg*, *nlamb=30*, *lamb_min_ratio=0.0001*, *lamb=None*, *scale_y=1*, *penalty_factor=None*, *reltol=1e-05*, *maxiter=1000*, *steps=1*, *btmax=100*, *c=0.0001*, *tau=2*, *M=4*, *nu=1*, *Lmin=0*, *lse=True*, *nthreads=4*)

Efficient procedure for solving the Lasso or SCAD penalized soft maximin problem.

This software implements two proximal gradient based algorithms (NPG and FISTA) to solve different forms of the soft maximin problem from Lund et al., 2022 see [1]. 1) For general group specific design the soft maximin problem is solved using the NPG algorithm. 2) For fixed identical design across groups, the estimation procedure uses either the FISTA algorithm or the NPG algorithm in the following two cases: i) For a tensor design matrix the algorithms use array arithmetic to avoid the design matrix and speed computations ii) For a wavelet based design matrix the algorithms use the pyramid algorithm to avoid the design matrix and speed up computations.

Multi-threading is possible when openMP is available.

Parameters

- **y** [list of arrays or array] For a model with varying design across groups a list containing the G group specific response vectors of sizes $n_i \times 1$. For a model with identical design across G groups, an array of size $n_1 \times \cdots \times n_d \times G$ ($d \in \{1, 2, 3\}$).
- **x** [list of arrays or string] For a model with varying design across groups a list containing the G group specific design matrices of sizes $n_i \times p_i$. For a model with identical design across G groups, either i) a list containing the $d \in \{1, 2, 3\}$ marginal design matrices (tensor components) or ii) a string indicating the type of wavelets to be used, see `pysmme.transforms.wt` for options.
- **zeta** [array of strictly positive floats] controls the soft maximin approximation accuracy. When `len(zeta) > 1` the procedure will distribute the computations using the `nthreads` parameter below when openMP is available.
- **penalty** [string] specifies the penalty type. Possible values are `lasso, scad`.
- **alg** [string] specifies the optimization algorithm. Possible values are `npg, fista`.
- **nlambda** [strictly positive int] The number of `lamb` values used when lamb is not specified.
- **lamb_min_ratio** [strictly positive float] controls minimum `lamb` values by setting the ratio bewtween λ_{max} – the (data dependent) smallest value for which all coefficients are zero – and the smallest value for `lamb`. Used when lamb is not specified.
- **lamb** [array of strictly positive floats] used as penalty parameters.
- **scale_y** [strictly positive float] that is the response `y` is multiplied with.
- **penalty_factor** [array] size $p_1 \times \cdots \times p_d$ of positive floats. Is multiplied with each element in `lamb` to allow differential penalization on the coefficients.
- **reltol** [strictly positive float] giving the convergence tolerance for the inner loop.
- **maxiter** [positive int] giving the maximum number of iterations allowed for each `lamb` value, when summing over all outer iterations for said `lamb`.
- **steps** [strictly positive int] giving the number of steps used in the multi-step adaptive lasso algorithm for non-convex penalties. Automatically set to 1 when `penalty = "lasso"`.
- **btmax** [strictly positive integer giving the maximum number of backtracking] steps allowed in each iteration.
- **c** [strictly positive float] used in the NPG algorithm.
- **tau** [strictly positive float] used to control the stepsize for NPG.
- **M** [pos int] giving the look back for the NPG.
- **nu** [strictly positive] float used to control the stepsize in the proximal algorithm. A value less than 1 will decrease the stepsize and a value larger than one will increase it.
- **Lmin** [pos float] used by the NPG algorithm to control the stepsize. For the default `Lmin = 0` the maximum step size is the same as for the FISTA algorithm.
- **lse** [bool] indicating whether to use log sum exp-loss. TRUE is default and yields the loss below.
- **nthreads** [pos int] giving the number of threads to use when openMP is available.

Returns

- **spec** [string] contains specifications of the model fitted by the function call

coef [array] A math:*p times`nlamb matrix containing the estimates of the model coefficients ( :math: `beta*) for each `lamb`-value for which the procedure converged. When `len(zeta) > 1` a `len(zeta)`-list of such matrices.

lamb [Array] containing the sequence of penalty values used in the estimation procedure for which the procedure converged. When `len(zeta) > 1` a `len(zeta)`-list of such vectors.

Obj [array] containing the objective values for each iteration and each model for which the procedure converged. When `len(zeta) > 1` a `len(zeta)`-list of such matrices.

df [array] Indicating the nonzero model coefficients for each value of `lamb` for which the procedure converged. When `len(zeta) > 1` a `len(zeta)`-list of such vectors.

dimcoef [int or np.array] Indicating the number p of model parameters. For array data a vector giving the dimension of the model coefficient array β

dimobs [integer] The number of observations. For array data a vector giving the number of observations in each dimension.

dimmodel [int] The dimension of the array model. `None` for general models.

iter [np.array] The number of iterations for each `lamb` value for which the procedure converged. When `len(zeta) > 1` a `len(zeta)`-list of such vectors. `bt_iter` is total number of backtracking steps performed, `bt_enter` is the number of times the backtracking is initiated, and `iter` is a vector containing the number of iterations for each `lamb` value and `iter` is total number of iterations. TODO!!! notoutputted

Notes

Consider modeling heterogeneous data $\{y_1, \ldots, y_n\}$ by dividing it into G groups $\mathbf{y}_g = (y_1, \ldots, y_{n_g})$, $g \in \{1, \ldots, G\}$ and then using a linear model

$$\mathbf{y}_g = \mathbf{X}_g b_g + \epsilon_g, \ g \in \{1, \ldots, G\},$$

to model the group response. Then b_g is a group specific $p \times 1$ coefficient vector, $\mathbf{X}_g$ an $n_g \times p$ group design matrix and ϵ_g an $n_g \times 1$ error term. The objective is to estimate a common coefficient β such that $\mathbf{X}_g\beta$ is a robust and good approximation to $\mathbf{X}_g b_g$ across groups.

Following [1], this objective may be accomplished by solving the soft maximin estimation problem

$$\min_{\beta} \frac{1}{\zeta} \log\left(\sum_{g=1}^{G} \exp(-\zeta \hat{V}_g(\beta))\right) + \lambda \|\beta\|_1, \quad \zeta > 0, \lambda \geq 0.$$

Here ζ essentially controls the amount of pooling across groups ($\zeta \sim 0$ effectively ignores grouping and pools observations) and

$$\hat{V}_g(\beta) := \frac{1}{n_g}(2\beta^\top \mathbf{X}_g^\top \mathbf{y}_g - \beta^\top \mathbf{X}_g^\top \mathbf{X}_g \beta),$$

is the empirical explained variance, see [2] for more details and references.

The function `softmaximin` solves the soft maximin estimation problem in large scale settings for a sequence of penalty parameters $\lambda_{max} > \ldots > \lambda_{min} > 0$ and a sequence of strictly positive softmaximin parameters $\zeta_1, \zeta_2, \ldots$.

The implementation also solves the problem above with the penalty given by the SCAD penalty, using the multiple step adaptive lasso procedure to loop over the inner proximal algorithm.

Two optimization algorithms are implemented in the SMME packages; a non-monotone proximal gradient (NPG) algorithm and a fast iterative soft thresholding algorithm (FISTA).

The implementation is particularly efficient for models where the design is identical across groups i.e. $\mathbf{X}_g = \mathbf{X}$ $\forall g \in \{1, \ldots, G\}$ in the following two cases:

i) first if $\mathbf{X}$ has Kronecker (tensor) structure i.e. for marginal $n_i \times p_i$ design matrices $\mathbf{M}_1, \ldots, \mathbf{M}_d$, $d \in \{1, 2, 3\}$,

$$\mathbf{X} = \bigotimes_{i=1}^{d} \mathbf{M}_i$$

then `y` is a $d+1$ dimensional response array and `x` is a list containing the d marginal matrices $\mathbf{M}_1, \ldots, \mathbf{M}_d$. In this case softmaximin solves the soft maximin problem using minimal memory by way of tensor optimized arithmetic, see also `RH`.

ii) second, if the design matrix $\mathbf{X}$ is the inverse matrix of an orthogonal wavelet transform then `softmaximin` will solve the soft maximin problem given `x = str` – where `str` is a shorthand for the wavelet basis (see. . . .) – and the $d+1$ dimensional response array `y`. In this case the pyramid algorithm is used to compute multiplications involving $\mathbf{X}$.

Note that when multiple values for ζ is provided it is possible to distribute the computations across CPUs if openMP is available.

References

[1]

Examples

#Non-array data ##size of example

```
>>> G = 3;
>>> n = np.array([65, 26, 13])
>>> p = np.array([13, 5, 4])
```

##marginal design matrices (Kronecker components)

```
>>> x = [None] * 3
>>> for i in range(len(x)):
>>> x[i] = np.random.normal(0, 1, (n[i], p[i]))
```

##common features and effects

```
>>> common_features = np.random.binomial(1, 0.1, np.prod(p)) #sparsity of common␣
↪effects
>>> common_effects = np.random.normal(size = np.prod(p)) * common_features
```

##group response

```
>>> y = np.zeros((n[0], n[1], n[2], G))
>>> for g in range(G):
>>>     bg = np.random.normal(0, 0.1, np.prod(p)) * (1 - common_features) + common_
↪effects
>>>     mu = RH(x[2], RH(x[1], RH(x[0], np.reshape(bg, (p[0], p[1], p[2]), "F") )))
>>>     y[:, :, :, g] = np.random.normal(0, 1, (n)) + mu
```

##fit model for range of lambda and zeta

```
>>> zeta = np.array([0.1, 1, 10, 100])
>>> fit = softmaximin(y, x, zeta = zeta, penalty = "lasso", alg = "npg")
>>> modelno = 10
>>> zetano = 2
>>> betahat = fit["coef"][zetano][:, modelno]
```

```
>>> f, ax = plt.subplots(1)
>>> ax.plot(common_effects, "r+")
>>> ax.plot(betahat)
>>> plt.show()
```

#Array data and wavelets ##size of example

```
>>> set.seed(42)
>>> G = 5;
>>> p = n = np.array([2**2, 2**3, 2**4])
```

##common features and effects

```
>>> common_features = np.random.binomial(1, 0.1, np.prod(p)) #sparsity of common␣
↪effects
>>> common_effects = np.random.normal(size = np.prod(p)) * common_features
```

##group response

```
>>> y = np.zeros((n[0], n[1], n[2], G))
>>> for g in range(G):
>>>     bg = np.random.normal(0, 0.1, np.prod(p)) * (1 - common_features) + common_
↪effects
>>>     mu = iwt(np.reshape(bg, (p[0], p[1], p[2]), "F"))
>>>     y[:, :, :, g] = np.random.normal(0, 1, (n)) + mu
```

##fit model for range of lambda and zeta

```
>>> zeta = np.array([0.1, 1, 10, 100])
>>> fit = softmaximin(y, x, zeta = zeta, penalty = "lasso", alg = "npg")
>>> modelno = 10
>>> zetano = 2
>>> betahat = fit["coef"][zetano][:, modelno]
```

```
>>> f, ax = plt.subplots(1)
>>> ax.plot(common_effects, "r+")
>>> ax.plot(betahat)
>>> plt.show()
```

##Non-array data ##size of example

```
>>> G = 10
>>> n = np.random.choice(np.arange(100,500,1), G) #sample(100:500, G);
>>> p = 60
>>> x = [None] * G
```

##group design matrices

```
>>> for i in range(len(x)):
>>> x[i] = np.random.normal(0, 1, (n[i], p))
```

##common features and effects

```
>>> common_features = np.random.binomial(1, 0.1, np.prod(p)) #sparsity of common␣
↪effects
>>> common_effects = np.random.normal(size = np.prod(p)) * common_features
```

##group response

```
>>> y = [None] * G
>>> for g in range(G):
>>>     bg = np.random.normal(0, 0.5, np.prod(p)) * (1 - common_features) + common_
↪effects
>>>     mu = np.matmul(x[g], bg)
>>>     y[g] = np.random.normal(0, 1, n[g]) + mu
```

##fit model for range of lamb and zeta

```
>>> fit = softmaximin(y, x, zeta = zeta, penalty = "lasso", alg = "npg")
>>> betahat = fit["coef"]
```

##estimated common effects for specific lamb and zeta

```
>>> modelno = 6
>>> zetano = 2
>>> f, ax = plt.subplots(1)
>>> ax.plot(common_effects, "r+")
>>> ax.plot(betahat[zetano][:, modelno])
>>> plt.show()
```

1.1.2 pysmme.transforms module

This module contains various transforms for computing fast matrix vector products in specific situations.

`pysmme.transforms.`**`H`**(*M*, *A*)

`pysmme.transforms.`**`RH`**(*M*, *A*)

The Rotated H-transform of a 3d Array by a Matrix.

This function is an implementation of the ρ-operator found in {Currie et al 2006}. It forms the basis of the GLAM arithmetic.

Parameters

M [np.array] A $n \times p_1$ matrix.

A [np.array] A 3d array of size $p_1 \times p_2 \times p_3$.

Returns

np.array A 3d array of size $p_2 \times p_3 \times n$.

Notes

For details see {Currie et al 2006} [2]. Note that this particular implementation is not used in the routines underlying the optimization procedure.

References

[2]

Examples

```
>>> n1 = 15; n2 = 4; n3 = 3; p1 = 12; p2 = 3; p3 = 4
>>> ###marginal design matrices (Kronecker components)
>>> X1 = np.random.normal(0, 1, (n1, p1))
>>> X2 = np.random.normal(0, 1, (n2, p2))
>>> X3 = np.random.normal(0, 1, (n3, p3))
>>> A = np.random.normal(0, 1, (p1, p2, p3))
>>> R1 = RH(X3, RH(X2, RH(X1, A)))
>>> R2 = np.matmul(np.kron(X3, np.kron(X2, X1)), np.reshape(A, [p1 * p2 * p3, 1], "F
↪"))
>>> max(abs(np.reshape(R1, [n1 * n2 * n3, 1], "F") - R2))
```

`pysmme.transforms.`**`Rotate`**(*A*)

`pysmme.transforms.`**`iwt`**(*x*, *wf='la8'*, *J=None*)

Discrete inverse wavelet transform.

This function performs a level J wavelet transform of the input array (1d, 2d, or 3d) using the pyramid algorithm (Mallat 1989). Implemented in C by Brandon Whithcer.

Parameters

x [np.array] a 1, 2, or 3 dimensional data array. The size of each dimension must be dyadic.

wf [string] the type of wavelet family used.

J [int] J is the level (depth) of the decomposition. For default None the max depth is used making `wt(x)` equal to multiplying **x** with the corresponding wavelet matrix.

Returns

np.array np.array of shape identical to input x continant the transformed

Notes

This is a C++/Python wrapper function for a C implementation of the discrete inverse wavelet transform. Given a data array (1d, 2d or 3d) with dyadic dimensions sizes this transform is computed efficiently via the pyramid algorithm using C routines from Brandon Whitcher's Waveslim package for R, see Percival and Walden (2000); Gencay, Selcuk and Whitcher (2001).

This functionality is used in the computations underlying `softmaximin` to perform multiplications involving the wavelet (design) matrix efficiently.

`pysmme.transforms.`**`wt`**(*x*, *wf='la8'*, *J=None*)

Discrete wavelet transform.

This function performs a level J wavelet transform of the input array (1d, 2d, or 3d) using the pyramid algorithm (Mallat 1989). Implemented in C by Brandon Whithcer.

Parameters

x [np.array] A 1, 2, or 3 dimensional data array. The size of each dimension must be dyadic.

wf [string] The type of wavelet family used.

J [int] J is the level (depth) of the decomposition. For default None the max depth is used and `wt(x)` is equal to multiplying `x` with the corresponding wavelet matrix.

Returns

np.array np.array of shape identical to input x continant the transformed x

Notes

This is a C++/Python wrapper function for a C implementation of the discrete wavelet transform. Given a data array (1d, 2d or 3d) with dyadic dimensions sizes this transform is computed efficiently via the pyramid algorithm using C routines from Brandon Whitcher's Waveslim package for R, see Percival and Walden (2000); Gencay, Selcuk and Whitcher (2001).

This functionality is used in the computations underlying {{softmaximin}} to perform multiplications involving the wavelet (design) matrix efficiently.

References

[3], [4], [5]

Examples

```
>>> d = np.reshape(np.arange(1,2**3 + 1,1), (2, 2, 2), order = "F")
>>> d1 = wt(d)
>>> d2 = np.array([[[ 1.41421356e+00,  4.16333634e-17],
      [ 5.65685425e+00, -3.33644647e-16]],
    [[ 2.82842712e+00, -2.77555756e-17],
      [-2.64953102e-16,  1.27279221e+01]]])
>>> iwt(d2)
```

1.1.3 Module contents

Root module of your package

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Lund, A., S. W. Mogensen and N. R. Hansen (2022). Soft Maximin Estimation for Heterogeneous Data. Scandinavian Journal of Statistics. url = {https://doi.org/10.1111/sjos.12580}

[2] Currie, I. D., M. Durban, and P. H. C. Eilers (2006). Generalized linear array models with applications to multidimensional smoothing. {Journal of the Royal Statistical Society. Series B}. 68, 259-280. url = {http://dx.doi.org/10.1111/j.1467-9868.2006.00543.x}.

[3] Gencay, R., F. Selcuk and B. Whitcher (2001) An Introduction to Wavelets and Other Filtering Methods in Finance and Economics, Academic Press.

[4] Mallat, S. G. (1989) A theory for multiresolution signal decomposition: the wavelet representation, IEEE Transactions on Pattern Analysis and Machine Intelligence, 11, No. 7, 674-693.

[5] Percival, D. B. and A. T. Walden (2000) Wavelet Methods for Time Series Analysis, Cambridge University Press.

PYTHON MODULE INDEX

INDEX