Package: pense (via r-universe)

October 17, 2026

Type Package

Title Penalized Elastic Net S/MM-Estimator of Regression

Version 2.5.2

Date 2026-01-26

Copyright See the file COPYRIGHT for copyright details on some of the functions and algorithms used.

Encoding UTF-8

Biarch true

URL https://dakep.github.io/pense-rpkg/, https://github.com/dakep/pense-rpkg

BugReports https://github.com/dakep/pense-rpkg/issues

Description Robust penalized (adaptive) elastic net S and M estimators for linear regression. The adaptive methods are proposed in Kepplinger, D. (2023) <doi:10.1016/j.csda.2023.107730> and the non-adaptive methods in Cohen Freue, G. V., Kepplinger, D., Salibián-Barrera, M., and Smucler, E. (2019) <doi:10.1214/19-AOAS1269>. The package implements robust hyper-parameter selection with robust information sharing cross-validation according to Kepplinger & Wei (2025) <doi:10.1080/00401706.2025.2540970>.

Depends R (>= 4.1.0), Matrix

Imports Rcpp, methods, parallel, rlang (>= 0.4.0)

LinkingTo Rcpp, RcppArmadillo (>= 0.9.600), testthat

Suggests testthat (>= 2.1.0), robustbase, knitr, rmarkdown, jsonlite, xml2

License MIT + file LICENSE

NeedsCompilation yes

RoxygenNote 7.3.3

VignetteBuilder knitr

Author David Kepplinger [aut, cre], Matías Salibián-Barrera [aut], Gabriela Cohen Freue [aut]

Maintainer David Kepplinger <david.kepplinger@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-01-27 11:40:09 UTC

RemoteUrl https://github.com/cran/pense

RemoteRef HEAD

RemoteSha 302f1a4deae69f32b0cd7801eaaa7367808cd90b

Contents

cd_algorithm_options . . . 3
change_cv_measure . . . 4
coef.pense_cvfit . . . 4
coef.pense_fit . . . 6
consistency_const . . . 8
elnet . . . 9
elnet_cv . . . 11
en_admm_options . . . 14
en_algorithm_options . . . 15
en_cd_options . . . 15
en_dal_options . . . 16
en_lars_options . . . 17
enpy_initial_estimates . . . 17
enpy_options . . . 19
mloc . . . 20
mlocscale . . . 21
mm_algorithm_options . . . 22
mscale . . . 22
mscale_algorithm_options . . . 23
pense . . . 24
pense_cv . . . 28
plot.pense_cvfit . . . 32
plot.pense_fit . . . 34
predict.pense_cvfit . . . 35
predict.pense_fit . . . 36
prediction_performance . . . 38
prinsens . . . 39
regmest . . . 40
regmest_cv . . . 43
residuals.pense_cvfit . . . 47
residuals.pense_fit . . . 48
rho_function . . . 50
starting_point . . . 50
summary.pense_cvfit . . . 52
tau_size . . . 53

Index **54**

`cd_algorithm_options` *Coordinate Descent (CD) Algorithm to Compute Penalized Elastic Net S-estimates*

Description

Set options for the CD algorithm to compute adaptive EN S-estimates.

Usage

```
cd_algorithm_options(
  max_it = 1000,
  reset_it = 8,
  linesearch_steps = 4,
  linesearch_mult = 0.5
)
```

Arguments

`max_it`	maximum number of iterations.
`reset_it`	number of iterations after which the residuals are re-computed from scratch, to prevent numerical drifts from incremental updates.
`linesearch_steps`	maximum number of steps used for line search.
`linesearch_mult`	multiplier to adjust the step size in the line search.

Value

options for the CD algorithm to compute (adaptive) PENSE estimates.

See Also

mm_algorithm_options to optimize the non-convex PENSE objective function via a sequence of convex problems.

Other Robust EN algorithms: `mm_algorithm_options()`

change_cv_measure *Change the Cross-Validation Measure*

Description

For cross-validated fits using the RIS-CV strategy, the measure of prediction accuracy can be adjusted post-hoc.

Usage

```
change_cv_measure(
  x,
  measure = c("wrmspe", "wmape", "tau_size", "wrmspe_cv", "wmape_cv"),
  max_solutions = Inf
)
```

Arguments

x	fitted (adaptive) PENSE or M-estimator
measure	the measure to use for prediction accuracy
max_solutions	consider only this many of the best solutions. If missing, all solutions are considered.

Value

a `pense.cvfit` object using the updated measure of prediction accuracy

See Also

Other functions to compute robust estimates with CV: `pense_cv()`, `regmest_cv()`

coef.pense_cvfit *Extract Coefficient Estimates*

Description

Extract coefficients from an adaptive PENSE (or LS-EN) regularization path with hyper-parameters chosen by cross-validation.

Usage

```
## S3 method for class 'pense_cvfit'
coef(
  object,
  alpha = NULL,
  lambda = "min",
  se_mult = 1,
  sparse = NULL,
  standardized = FALSE,
  ...
)
```

Arguments

`object`	PENSE with cross-validated hyper-parameters to extract coefficients from.
`alpha`	Either a single number or `NULL` (default). If given, only fits with the given `alpha` value are considered. If `lambda` is a numeric value and `object` was fit with multiple *alpha* values and no value is provided, the first value in `object$alpha` is used with a warning.
`lambda`	either a string specifying which penalty level to use (`"min"`, `"se"`, `"{m}-se"`) or a single numeric value of the penalty parameter. See details.
`se_mult`	If `lambda = "se"`, the multiple of standard errors to tolerate.
`sparse`	should coefficients be returned as sparse or dense vectors? Defaults to the sparsity setting of the given `object`. Can also be set to `sparse = 'matrix'`, in which case a sparse matrix is returned instead of a sparse vector.
`standardized`	return the standardized coefficients.
`...`	currently not used.

Value

either a numeric vector or a sparse vector of type dsparseVector of size $p + 1$, depending on the `sparse` argument. Note: prior to version 2.0.0 sparse coefficients were returned as sparse matrix of type *dgCMatrix*. To get a sparse matrix as in previous versions, use `sparse = 'matrix'`.

Hyper-parameters

If `lambda = "{m}-se"` and `object` contains fitted estimates for every penalization level in the sequence, use the fit the most parsimonious model with prediction performance statistically indistinguishable from the best model. This is determined to be the model with prediction performance within `m * cv_se` from the best model. If `lambda = "se"`, the multiplier *m* is taken from `se_mult`.

By default all *alpha* hyper-parameters available in the fitted object are considered. This can be overridden by supplying one or multiple values in parameter `alpha`. For example, if `lambda = "1-se"` and `alpha` contains two values, the "1-SE" rule is applied individually for each `alpha` value, and the fit with the better prediction error is considered.

In case `lambda` is a number and `object` was fit for several *alpha* hyper-parameters, `alpha` must also be given, or the first value in `object$alpha` is used with a warning.

See Also

Other functions for extracting components: `coef.pense_fit()`, `predict.pense_cvfit()`, `predict.pense_fit()`, `residuals.pense_cvfit()`, `residuals.pense_fit()`

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- pense(x, freeny$y, alpha = 0.5)
plot(regpath)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- pense_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

---

coef.pense_fit *Extract Coefficient Estimates*

Description

Extract coefficients from an adaptive PENSE (or LS-EN) regularization path fitted by `pense()` or `elnet()`.

Usage

```
## S3 method for class 'pense_fit'
coef(object, lambda, alpha = NULL, sparse = NULL, standardized = FALSE, ...)
```

Arguments

`object`	PENSE regularization path to extract coefficients from.
`lambda`	a single number for the penalty level.
`alpha`	Either a single number or `NULL` (default). If given, only fits with the given `alpha` value are considered. If `object` was fit with multiple `alpha` values, and no value is provided, the first value in `object$alpha` is used with a warning.
`sparse`	should coefficients be returned as sparse or dense vectors? Defaults to the sparsity setting in `object`. Can also be set to `sparse = 'matrix'`, in which case a sparse matrix is returned instead of a sparse vector.
`standardized`	return the standardized coefficients.
`...`	currently not used.

Value

either a numeric vector or a sparse vector of type dsparseVector of size $p + 1$, depending on the `sparse` argument. Note: prior to version 2.0.0 sparse coefficients were returned as sparse matrix of type *dgCMatrix*. To get a sparse matrix as in previous versions, use `sparse = 'matrix'`.

See Also

`coef.pense_cvfit()` for extracting coefficients from a PENSE fit with hyper-parameters chosen by cross-validation

Other functions for extracting components: `coef.pense_cvfit()`, `predict.pense_cvfit()`, `predict.pense_fit()`, `residuals.pense_cvfit()`, `residuals.pense_fit()`

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- pense(x, freeny$y, alpha = 0.5)
plot(regpath)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- pense_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
```

```
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

`consistency_const` *Get the Constant for Consistency for the M-Scale and for Efficiency for the M-estimate of Location*

Description

Returns the tuning constants required to achieve the desired breakdown point or efficiency under the Normal model.

Usage

```
consistency_const(delta, rho, eps = sqrt(.Machine$double.eps))

efficiency_const(eff, rho, eps = sqrt(.Machine$double.eps))
```

Arguments

`delta`	desired breakdown point (between 0 and 0.5)
`rho`	the name of the chosen ρ function. See `rho_function()` for a list of supported functions.
`eps`	numerical tolerance level for equality comparisons
`eff`	desired asymptotic efficiency (between 0.1 and 0.99).

Value

consistency constant

See Also

Other Robustness control options: `mscale_algorithm_options()`, `rho_function()`

Other Robustness control options: `mscale_algorithm_options()`, `rho_function()`

elnet *Compute the Least Squares (Adaptive) Elastic Net Regularization Path*

Description

Compute least squares EN estimates for linear regression with optional observation weights and penalty loadings.

Usage

```
elnet(
  x,
  y,
  alpha,
  nlambda = 100,
  lambda_min_ratio,
  lambda,
  penalty_loadings,
  weights,
  intercept = TRUE,
  en_algorithm_opts,
  sparse = FALSE,
  eps = 1e-06,
  standardize = TRUE
)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n. For binary classification, y should be a factor with 2 levels.
alpha	elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. alpha = 1 is the LASSO penalty, and alpha = 0 the Ridge penalty. Can be a vector of several values, but alpha = 0 cannot be mixed with other values.
nlambda	number of penalization levels.
lambda_min_ratio	Smallest value of the penalization level as a fraction of the largest level (i.e., the smallest value for which all coefficients are zero). The default depends on the sample size relative to the number of variables and alpha. If more observations than variables are available, the default is 1e-3 * alpha, otherwise 1e-2 * alpha.
lambda	optional user-supplied sequence of penalization levels. If given and not NULL, nlambda and lambda_min_ratio are ignored.
penalty_loadings	a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient.

`weights` a vector of positive observation weights.

`intercept` include an intercept in the model.

`en_algorithm_opts` options for the EN algorithm. See en_algorithm_options for details.

`sparse` use sparse coefficient vectors.

`eps` numerical tolerance.

`standardize` standardize variables to have unit variance. Coefficients are always returned in original scale.

Details

The elastic net estimator for the linear regression model solves the optimization problem

$$argmin_{\mu,\beta}(1/2n)\sum_i w_i(y_i - \mu - x_i'\beta)^2 + \lambda\sum_j 0.5(1-\alpha)\beta_j^2 + \alpha l_j|\beta_j|$$

with observation weights w_i and penalty loadings l_j.

Value

a list-like object with the following items

`alpha` the sequence of `alpha` parameters.

`lambda` a list of sequences of penalization levels, one per `alpha` parameter.

`estimates` a list of estimates. Each estimate contains the following information:

- `intercept` intercept estimate.
- `beta` beta (slope) estimate.
- `lambda` penalization level at which the estimate is computed.
- `alpha` *alpha* hyper-parameter at which the estimate is computed.
- `statuscode` if `> 0` the algorithm experienced issues when computing the estimate.
- `status` optional status message from the algorithm.

`call` the original call.

See Also

`pense()` for an S-estimate of regression with elastic net penalty.

`coef.pense_fit()` for extracting coefficient estimates.

`plot.pense_fit()` for plotting the regularization path.

Other functions for computing non-robust estimates: `elnet_cv()`

Examples

```
# Compute the LS-EN regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- elnet(x, freeny$y, alpha = c(0.5, 0.75))
plot(regpath)
plot(regpath, alpha = 0.75)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[5]],
     alpha = 0.75)

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- elnet_cv(x, freeny$y, alpha = c(0.5, 0.75),
                       cv_repl = 10, cv_k = 4,
                       cv_measure = "tau")
plot(cv_results, se_mult = 1.5)
plot(cv_results, se_mult = 1.5, what = "coef.path")


# Extract the coefficients at the penalization level with
# smallest prediction error ...
summary(cv_results)
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
summary(cv_results, lambda = "1.5-se")
coef(cv_results, lambda = "1.5-se")
```

`elnet_cv` *Cross-validation for Least-Squares (Adaptive) Elastic Net Estimates*

Description

Perform (repeated) K-fold cross-validation for `elnet()`.

Usage

```
elnet_cv(
  x,
  y,
  lambda,
  cv_k,
  cv_repl = 1,
  cv_type = "naive",
  cv_metric = c("rmspe", "tau_size", "mape", "auroc"),
```

```
  fit_all = TRUE,
  cl = NULL,
  ...
)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n. For binary classification, y should be a factor with 2 levels.
lambda	optional user-supplied sequence of penalization levels. If given and not NULL, nlambda and lambda_min_ratio are ignored.
cv_k	number of folds per cross-validation.
cv_repl	number of cross-validation replications.
cv_type	what kind of cross-validation should be performed: robust information sharing (ris) or standard (naive) CV.
cv_metric	only for cv_type='naive'. Either a string specifying the performance metric to use, or a function to evaluate prediction errors in a single CV replication. If a function, the number of arguments define the data the function receives. If the function takes a single argument, it is called with a single numeric vector of prediction errors. If the function takes two or more arguments, it is called with the predicted values as first argument and the true values as second argument. The function must always return a single numeric value quantifying the prediction performance. The order of the given values corresponds to the order in the input data.
fit_all	only for cv_type='naive'. If TRUE, fit the model for all penalization levels. Can also be any combination of "min" and "{x}-se", in which case only models at the penalization level with smallest average CV accuracy, or within {x} standard errors, respectively. Setting fit_all to FALSE is equivalent to "min". Applies to all alpha value.
cl	a parallel cluster. Can only be used in combination with ncores = 1.
...	Arguments passed on to elnet alpha elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. alpha = 1 is the LASSO penalty, and alpha = 0 the Ridge penalty. Can be a vector of several values, but alpha = 0 cannot be mixed with other values. nlambda number of penalization levels. lambda_min_ratio Smallest value of the penalization level as a fraction of the largest level (i.e., the smallest value for which all coefficients are zero). The default depends on the sample size relative to the number of variables and alpha. If more observations than variables are available, the default is 1e-3 * alpha, otherwise 1e-2 * alpha. penalty_loadings a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. standardize standardize variables to have unit variance. Coefficients are always returned in original scale.

`weights` a vector of positive observation weights.

`intercept` include an intercept in the model.

`sparse` use sparse coefficient vectors.

`en_algorithm_opts` options for the EN algorithm. See en_algorithm_options for details.

`eps` numerical tolerance.

Details

The built-in CV metrics are

`"tau_size"` τ-size of the prediction error, computed by `tau_size()` (default).

`"mape"` Median absolute prediction error.

`"rmspe"` Root mean squared prediction error.

`"auroc"` Area under the receiver operator characteristic curve (actually 1 - AUROC). Only sensible for binary responses.

Value

a list-like object with the same components as returned by `elnet()`, plus the following:

`cvres` data frame of average cross-validated performance.

See Also

`elnet()` for computing the LS-EN regularization path without cross-validation.

`pense_cv()` for cross-validation of S-estimates of regression with elastic net penalty.

`coef.pense_cvfit()` for extracting coefficient estimates.

`plot.pense_cvfit()` for plotting the CV performance or the regularization path.

Other functions for computing non-robust estimates: `elnet()`

Examples

```
# Compute the LS-EN regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- elnet(x, freeny$y, alpha = c(0.5, 0.75))
plot(regpath)
plot(regpath, alpha = 0.75)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[5]],
     alpha = 0.75)

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- elnet_cv(x, freeny$y, alpha = c(0.5, 0.75),
```

```
                        cv_repl = 10, cv_k = 4,
                        cv_measure = "tau")
plot(cv_results, se_mult = 1.5)
plot(cv_results, se_mult = 1.5, what = "coef.path")


# Extract the coefficients at the penalization level with
# smallest prediction error ...
summary(cv_results)
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
summary(cv_results, lambda = "1.5-se")
coef(cv_results, lambda = "1.5-se")
```

`en_admm_options` *Use the ADMM Elastic Net Algorithm*

Description

Use the ADMM Elastic Net Algorithm

Usage

```
en_admm_options(max_it = 1000, step_size, acceleration = 1)
```

Arguments

`max_it`	maximum number of iterations.
`step_size`	step size for the algorithm.
`acceleration`	acceleration factor for linearized ADMM.

Value

options for the ADMM EN algorithm.

See Also

Other LS-EN algorithm options: `en_algorithm_options`, `en_cd_options()`, `en_dal_options()`, `en_lars_options()`

en_algorithm_options *Control the Algorithm to Compute (Weighted) Least-Squares Elastic Net Estimates*

Description

The package supports different algorithms to compute the EN estimate for weighted LS loss functions. Each algorithm has certain characteristics that make it useful for some problems. To select a specific algorithm and adjust the options, use any of the `en_***_options` functions.

Details

- `en_lars_options()`: Use the tuning-free LARS algorithm. This computes *exact* (up to numerical errors) solutions to the EN-LS problem. It is not iterative and therefore can not benefit from approximate solutions, but in turn guarantees that a solution will be found.
- `en_cd_options()`: Use an iterative coordinate descent algorithm which needs $O(np)$ operations per iteration and converges sub-linearly.
- `en_admm_options()`: Use an iterative ADMM-type algorithm which needs $O(np)$ operations per iteration and converges sub-linearly.
- `en_dal_options()`: Use the iterative Dual Augmented Lagrangian (DAL) method. DAL needs $O(n^3p^2)$ operations per iteration, but converges exponentially.

See Also

Other LS-EN algorithm options: `en_admm_options()`, `en_cd_options()`, `en_dal_options()`, `en_lars_options()`

en_cd_options *Use Coordinate Descent to Solve Elastic Net Problems*

Description

Use Coordinate Descent to Solve Elastic Net Problems

Usage

```
en_cd_options(max_it = 1000, reset_it = 8)
```

Arguments

`max_it`	maximum number of iterations.
`reset_it`	number of iterations after which the residuals are re-computed from scratch, to prevent numerical drifts from incremental updates.

See Also

Other LS-EN algorithm options: `en_admm_options()`, `en_algorithm_options`, `en_dal_options()`, `en_lars_options()`

`en_dal_options` *Use the DAL Elastic Net Algorithm*

Description

Use the DAL Elastic Net Algorithm

Usage

```
en_dal_options(
  max_it = 100,
  max_inner_it = 100,
  eta_multiplier = 2,
  eta_start_conservative = 0.01,
  eta_start_aggressive = 1,
  lambda_relchange_aggressive = 0.25
)
```

Arguments

`max_it`	maximum number of (outer) iterations.
`max_inner_it`	maximum number of (inner) iterations in each outer iteration.
`eta_multiplier`	multiplier for the barrier parameter. In each iteration, the barrier must be more restrictive (i.e., the multiplier must be > 1).
`eta_start_conservative`	conservative initial barrier parameter. This is used if the previous penalty is undefined or too far away.
`eta_start_aggressive`	aggressive initial barrier parameter. This is used if the previous penalty is close.
`lambda_relchange_aggressive`	how close must the lambda parameter from the previous penalty term be to use an aggressive initial barrier parameter (i.e., what constitutes "too far").

Value

options for the DAL EN algorithm.

See Also

Other LS-EN algorithm options: `en_admm_options()`, `en_algorithm_options`, `en_cd_options()`, `en_lars_options()`

en_lars_options *Use the LARS Elastic Net Algorithm*

Description

Use the LARS Elastic Net Algorithm

Usage

```
en_lars_options()
```

See Also

Other LS-EN algorithm options: en_admm_options(), en_algorithm_options, en_cd_options(), en_dal_options()

enpy_initial_estimates *ENPY Initial Estimates for EN S-Estimators*

Description

Compute initial estimates for the EN S-estimator using the EN-PY procedure.

Usage

```
enpy_initial_estimates(
  x,
  y,
  alpha,
  lambda,
  bdp = 0.25,
  cc,
  intercept = TRUE,
  penalty_loadings,
  enpy_opts = enpy_options(),
  mscale_opts = mscale_algorithm_options(),
  eps = 1e-06,
  sparse = FALSE,
  ncores = 1L
)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n.
alpha	elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. alpha = 1 is the LASSO penalty, and alpha = 0 the Ridge penalty. Can be a vector of several values, but alpha = 0 cannot be mixed with other values.
lambda	a vector of positive values of penalization levels.
bdp	desired breakdown point of the estimator, between 0.05 and 0.5. The actual breakdown point may be slightly larger/smaller to avoid instabilities of the S-loss.
cc	cutoff value for the rho function. By default, chosen to yield a consistent estimate for the Normal distribution.
intercept	include an intercept in the model.
penalty_loadings	a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. Only allowed for alpha > 0.
enpy_opts	options for the EN-PY algorithm, created with the enpy_options() function.
mscale_opts	options for the M-scale estimation. See mscale_algorithm_options() for details.
eps	numerical tolerance.
sparse	use sparse coefficient vectors.
ncores	number of CPU cores to use in parallel. By default, only one CPU core is used. Not supported on all platforms, in which case a warning is given.

Details

If these manually computed initial estimates are intended as starting points for pense(), they are by default *shared* for all penalization levels. To restrict the use of the initial estimates to the penalty level they were computed for, use as_starting_point(..., specific = TRUE). See as_starting_point() for details.

References

Cohen Freue, G.V.; Kepplinger, D.; Salibián-Barrera, M.; Smucler, E. Robust elastic net estimators for variable selection and identification of proteomic biomarkers. *Ann. Appl. Stat.* **13** (2019), no. 4, 2065–2090 doi:10.1214/19AOAS1269

See Also

Other functions for initial estimates: enpy_options(), prinsens(), starting_point()

enpy_options *Options for the ENPY Algorithm*

Description

Additional control options for the elastic net Peña-Yohai procedure.

Usage

```
enpy_options(
  max_it = 10,
  keep_psc_proportion = 0.5,
  en_algorithm_opts,
  keep_residuals_measure = c("threshold", "proportion"),
  keep_residuals_proportion = 0.5,
  keep_residuals_threshold = 2,
  retain_best_factor = 2,
  retain_max = 500
)
```

Arguments

`max_it`	maximum number of EN-PY iterations.
`keep_psc_proportion`	how many observations should to keep based on the Principal Sensitivity Components.
`en_algorithm_opts`	options for the LS-EN algorithm. See en_algorithm_options for details.
`keep_residuals_measure`	how to determine what observations to keep, based on their residuals. If `proportion`, a fixed number of observations is kept. If `threshold`, only observations with residuals below the threshold are kept.
`keep_residuals_proportion`	proportion of observations to kept based on their residuals.
`keep_residuals_threshold`	only observations with (standardized) residuals less than this threshold are kept.
`retain_best_factor`	only keep candidates that are within this factor of the best candidate. If `<= 1`, only keep candidates from the last iteration.
`retain_max`	maximum number of candidates, i.e., only the best `retain_max` candidates are retained.

Details

The EN-PY procedure for computing initial estimates iteratively cleans the data of observations with possibly outlying residual or high leverage. Least-squares elastic net (LS-EN) estimates are computed on the possibly clean subsets. At each iteration, the Principal Sensitivity Components are computed to remove observations with potentially high leverage. Among all the LS-EN estimates, the estimate with smallest M-scale of the residuals is selected. Observations with largest residual for the selected estimate are removed and the next iteration is started.

Value

options for the ENPY algorithm.

See Also

Other functions for initial estimates: `enpy_initial_estimates()`, `prinsens()`, `starting_point()`

`mloc` *Compute the M-estimate of Location*

Description

Compute the M-estimate of location using an auxiliary estimate of the scale.

Usage

```
mloc(x, scale, rho = "bisquare", eff = 0.9, cc, max_it = 200, eps = 1e-08)
```

Arguments

`x`	numeric values. Missing values are verbosely ignored.
`scale`	scale of the `x` values. If omitted, uses the mad().
`rho`	the ρ function to use. See `rho_function()` for available functions.
`eff`	desired efficiency under the Normal model.
`cc`	value of the tuning constant for the chosen ρ function. If specified, overrides the desired efficiency.
`max_it`	maximum number of iterations.
`eps`	numerical tolerance to check for convergence.

Value

a single numeric value, the M-estimate of location.

See Also

Other functions to compute robust estimates of location and scale: `mlocscale()`, `mscale()`, `tau_size()`

`mlocscale` *Compute the M-estimate of Location and Scale*

Description

Simultaneous estimation of the location and scale by means of M-estimates.

Usage

```
mlocscale(
  x,
  bdp = 0.25,
  eff = 0.9,
  scale_cc,
  location_rho,
  location_cc,
  opts = mscale_algorithm_options()
)
```

Arguments

`x`	numeric values. Missing values are verbosely ignored.
`bdp`	desired breakdown point (between 0 and 0.5).
`eff`	desired efficiency of the location estimate (between 0.1 and 0.99).
`scale_cc`	tuning constant for the ρ function for computing the scale estimate. By default, chosen to yield a consistent estimate for normally distributed values.
`location_rho`	ρ function for computing the location estimate. If missing, use the same function as for the scale estimate (`opts$rho`). See `rho_function()` for a list of available ρ functions.
`location_cc`	tuning constant for the location ρ function. By default chosen to yield the desired efficiency. If this is provided, the desired efficiency is ignored.
`opts`	a list of options for the M-scale estimating equations, See `mscale_algorithm_options()` for details.

Value

a vector with 2 elements, the M-estimate of location and the M-scale estimate.

See Also

Other functions to compute robust estimates of location and scale: `mloc()`, `mscale()`, `tau_size()`

mm_algorithm_options *MM-Algorithm to Compute Penalized Elastic Net S- and M-Estimates*

Description

Additional options for the MM algorithm to compute EN S- and M-estimates.

Usage

```
mm_algorithm_options(
  max_it = 500,
  tightening = c("adaptive", "exponential", "none"),
  tightening_steps = 2,
  en_algorithm_opts
)
```

Arguments

max_it	maximum number of iterations.
tightening	how to make inner iterations more precise as the algorithm approaches a local minimum.
tightening_steps	for *adaptive* tightening strategy, how often to tighten until the desired tolerance is attained.
en_algorithm_opts	options for the inner LS-EN algorithm. See en_algorithm_options for details.

Value

options for the MM algorithm.

See Also

cd_algorithm_options for a direct optimization of the non-convex PENSE loss.

Other Robust EN algorithms: cd_algorithm_options()

mscale *Compute the M-Scale of Centered Values*

Description

Compute the M-scale without centering the values.

Usage

```
mscale(x, bdp = 0.25, cc, opts = mscale_algorithm_options())
```

Arguments

x	numeric values. Missing values are verbosely ignored.
bdp	desired breakdown point (between 0 and 0.5).
cc	tuning parameters for the chosen rho function. By default, chosen to yield a consistent estimate for the Normal distribution.
opts	a list of options for the M-scale estimation algorithm, see `mscale_algorithm_options()` for details.

Value

the M-estimate of scale.

See Also

Other functions to compute robust estimates of location and scale: `mloc()`, `mlocscale()`, `tau_size()`

`mscale_algorithm_options` *Options for the M-scale Estimation Algorithm*

Description

Options for the M-scale Estimation Algorithm

Usage

```
mscale_algorithm_options(rho = "bisquare", max_it = 200, eps = 1e-08)
```

Arguments

rho	the ρ function to use. See `rho_function()` for possible values.
max_it	maximum number of iterations.
eps	numerical tolerance to check for convergence.

Value

options for the M-scale estimation algorithm.

See Also

Other Robustness control options: `consistency_const()`, `rho_function()`

pense *Compute (Adaptive) Elastic Net S-Estimates of Regression*

Description

Compute elastic net S-estimates (PENSE estimates) along a grid of penalization levels with optional penalty loadings for adaptive elastic net.

Usage

```
pense(
  x,
  y,
  alpha,
  nlambda = 50,
  nlambda_enpy = 10,
  lambda,
  lambda_min_ratio,
  enpy_lambda,
  penalty_loadings,
  intercept = TRUE,
  bdp = 0.25,
  cc,
  add_zero_based = TRUE,
  enpy_specific = FALSE,
  other_starts,
  carry_forward = TRUE,
  eps = 1e-06,
  explore_solutions = 0,
  explore_tol = 0.1,
  explore_it = 5,
  max_solutions = 5,
  comparison_tol = sqrt(eps),
  sparse = FALSE,
  ncores = 1,
  standardize = TRUE,
  algorithm_opts = mm_algorithm_options(),
  mscale_opts = mscale_algorithm_options(),
  enpy_opts = enpy_options(),
  ...
)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n. For binary classification, y should be a factor with 2 levels.

`alpha` elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. `alpha = 1` is the LASSO penalty, and `alpha = 0` the Ridge penalty. Can be a vector of several values, but `alpha = 0` cannot be mixed with other values.

`nlambda` number of penalization levels.

`nlambda_enpy` number of penalization levels where the EN-PY initial estimate is computed.

`lambda` optional user-supplied sequence of penalization levels. If given and not `NULL`, `nlambda` and `lambda_min_ratio` are ignored.

`lambda_min_ratio` Smallest value of the penalization level as a fraction of the largest level (i.e., the smallest value for which all coefficients are zero). The default depends on the sample size relative to the number of variables and `alpha`. If more observations than variables are available, the default is `1e-3 * alpha`, otherwise `1e-2 * alpha`.

`enpy_lambda` optional user-supplied sequence of penalization levels at which EN-PY initial estimates are computed. If given and not `NULL`, `nlambda_enpy` is ignored.

`penalty_loadings` a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. Only allowed for `alpha` > 0.

`intercept` include an intercept in the model.

`bdp` desired breakdown point of the estimator, between 0.05 and 0.5. The actual breakdown point may be slightly larger/smaller to avoid instabilities of the S-loss.

`cc` tuning constant for the S-estimator. Default is chosen based on the breakdown point `bdp`. This affects the estimated coefficients only if `standardize=TRUE`. Otherwise only the estimated scale of the residuals would be affected.

`add_zero_based` also consider the 0-based regularization path. See details for a description.

`enpy_specific` use the EN-PY initial estimates only at the penalization level they are computed for. See details for a description.

`other_starts` a list of other staring points, created by `starting_point()`. If the output of `enpy_initial_estimates()` is given, the starting points will be *shared* among all penalization levels. Note that if a the starting point is *specific* to a penalization level, this penalization level is added to the grid of penalization levels (either the manually specified grid in `lambda` or the automatically generated grid of size `nlambda`). If `standardize = TRUE`, the starting points are also scaled.

`carry_forward` carry the best solutions forward to the next penalty level.

`eps` numerical tolerance.

`explore_solutions` number of solutions to keep after the exploration step. The best `explore_solutions` are then iterated to full numerical tolerance `eps`. If 0, all non-duplicated solutions are kept.

`explore_tol`, `explore_it` numerical tolerance and maximum number of iterations for exploring possible solutions. The tolerance should be (much) looser than `eps` to be useful, and the number of iterations should also be much smaller than the maximum number of iterations given via `algorithm_opts`. `explore_tol` is also used to determine if two solutions are equal in the exploration stage.

`max_solutions`	retain only up to `max_solutions` unique solutions per penalization level.
`comparison_tol`	numeric tolerance to determine if two solutions are equal. The comparison is first done on the absolute difference in the value of the objective function at the solution. If this is less than `comparison_tol`, two solutions are deemed equal if the squared difference of the intercepts is less than `comparison_tol` and the squared L_2 norm of the difference vector is less than `comparison_tol`.
`sparse`	use sparse coefficient vectors.
`ncores`	number of CPU cores to use in parallel. By default, only one CPU core is used. Not supported on all platforms, in which case a warning is given.
`standardize`	logical flag to standardize the `x` variables prior to fitting the PENSE estimates. Coefficients are always returned on the original scale. This can fail for variables with a large proportion of a single value (e.g., zero-inflated data). In this case, either compute with `standardize = FALSE` or standardize the data manually.
`algorithm_opts`	options for the MM algorithm to compute the estimates. See `mm_algorithm_options()` for details.
`mscale_opts`	options for the M-scale estimation. See `mscale_algorithm_options()` for details.
`enpy_opts`	options for the ENPY initial estimates, created with the `enpy_options()` function. See `enpy_initial_estimates()` for details.
`...`	ignored.

Value

a list-like object with the following items

`alpha` the sequence of `alpha` parameters.

`lambda` a list of sequences of penalization levels, one per `alpha` parameter.

`estimates` a list of estimates. Each estimate contains the following information:

- `intercept` intercept estimate.
- `beta` beta (slope) estimate.
- `lambda` penalization level at which the estimate is computed.
- `alpha` *alpha* hyper-parameter at which the estimate is computed.
- `bdp` chosen breakdown-point.
- `objf_value` value of the objective function at the solution.
- `statuscode` if `> 0` the algorithm experienced issues when computing the estimate.
- `status` optional status message from the algorithm.

`bdp` the actual breakdown point used.

`call` the original call.

Strategies for Using Starting Points

The function supports several different strategies to compute, and use the provided starting points for optimizing the PENSE objective function.

Starting points are computed internally but can also be supplied via `other_starts`. By default, starting points are computed internally by the EN-PY procedure for penalization levels supplied in `enpy_lambda` (or the automatically generated grid of length `nlambda_enpy`). By default, starting points computed by the EN-PY procedure are *shared* for all penalization levels in `lambda` (or the automatically generated grid of length `nlambda`). If the starting points should be *specific* to the penalization level the starting points' penalization level, set the `enpy_specific` argument to `TRUE`.

In addition to EN-PY initial estimates, the algorithm can also use the "0-based" strategy if `add_zero_based = TRUE` (by default). Here, the 0-vector is used to start the optimization at the largest penalization level in `lambda`. At subsequent penalization levels, the solution at the previous penalization level is also used as starting point.

At every penalization level, all starting points are explored using the loose numerical tolerance `explore_tol`. Only the best `explore_solutions` are computed to the stringent numerical tolerance `eps`. Finally, only the best `max_solutions` are retained and carried forward as starting points for the subsequent penalization level.

See Also

`pense_cv()` for selecting hyper-parameters via cross-validation.

`coef.pense_fit()` for extracting coefficient estimates.

`plot.pense_fit()` for plotting the regularization path.

Other functions to compute robust estimates: `regmest()`

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- pense(x, freeny$y, alpha = 0.5)
plot(regpath)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- pense_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

pense_cv *Cross-validation for (Adaptive) PENSE Estimates*

Description

Perform (repeated) K-fold cross-validation for `pense()`.

`adapense_cv()` is a convenience wrapper to compute adaptive PENSE estimates.

Usage

```
pense_cv(
  x,
  y,
  standardize = TRUE,
  lambda,
  cv_k,
  cv_repl = 1,
  cv_type = c("ris", "naive"),
  cv_metric = c("tau_size", "mape", "rmspe", "auroc"),
  ris_min_similarity = 0.5,
  fit_all = TRUE,
  fold_starts = c("full", "enpy", "both"),
  cv_algorithm_opts,
  cl = NULL,
  ...
)

adapense_cv(x, y, alpha, alpha_preliminary = 0, exponent = 1, ...)
```

Arguments

`x`	`n` by `p` matrix of numeric predictors.
`y`	vector of response values of length `n`. For binary classification, `y` should be a factor with 2 levels.
`standardize`	whether to standardize the `x` variables prior to fitting the PENSE estimates. Can also be set to `"cv_only"`, in which case the input data is not standardized, but the training data in the CV folds is scaled to match the scaling of the input data. Coefficients are always returned on the original scale. This can fail for variables with a large proportion of a single value (e.g., zero-inflated data). In this case, either compute with `standardize = FALSE` or standardize the data manually.
`lambda`	optional user-supplied sequence of penalization levels. If given and not `NULL`, `nlambda` and `lambda_min_ratio` are ignored.
`cv_k`	number of folds per cross-validation.
`cv_repl`	number of cross-validation replications.

`cv_type` what kind of cross-validation should be performed: robust information sharing (`ris`) or standard (`naive`) CV.

`cv_metric` only for `cv_type='naive'`. Either a string specifying the performance metric to use, or a function to evaluate prediction errors in a single CV replication. If a function, the number of arguments define the data the function receives. If the function takes a single argument, it is called with a single numeric vector of prediction errors. If the function takes two or more arguments, it is called with the predicted values as first argument and the true values as second argument. The function must always return a single numeric value quantifying the prediction performance. The order of the given values corresponds to the order in the input data.

`ris_min_similarity` minimum average similarity of the CV solutions to be considered (between 0 and 1). If no CV solution satisfies this lower bound, the best CV solution will be used regardless of similarity.

`fit_all` only for `cv_type='naive'`. If `TRUE`, fit the model for all penalization levels. Can also be any combination of `"min"` and `"{x}-se"`, in which case only models at the penalization level with smallest average CV accuracy, or within `{x}` standard errors, respectively. Setting `fit_all` to `FALSE` is equivalent to `"min"`. Applies to all `alpha` value.

`fold_starts` how to determine starting values in the cross-validation folds. If `"full"` (default), use the best solution from the fit to the full data as starting value. This implies `fit_all=TRUE`. If `"enpy"` compute separate ENPY initial estimates in each fold. The option `"both"` uses both. These starts are in addition to the starts provided in `other_starts`.

`cv_algorithm_opts` Override algorithm options for the CV iterations. This is usually not necessary, unless the user wants to change the number of solutions retained for the CV training data.

`cl` a parallel cluster. Can only be used in combination with `ncores = 1`.

`...` Arguments passed on to `pense`

`nlambda` number of penalization levels.

`lambda_min_ratio` Smallest value of the penalization level as a fraction of the largest level (i.e., the smallest value for which all coefficients are zero). The default depends on the sample size relative to the number of variables and `alpha`. If more observations than variables are available, the default is `1e-3 * alpha`, otherwise `1e-2 * alpha`.

`nlambda_enpy` number of penalization levels where the EN-PY initial estimate is computed.

`penalty_loadings` a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. Only allowed for `alpha` > 0.

`enpy_lambda` optional user-supplied sequence of penalization levels at which EN-PY initial estimates are computed. If given and not `NULL`, `nlambda_enpy` is ignored.

`other_starts` a list of other staring points, created by `starting_point()`. If the output of `enpy_initial_estimates()` is given, the starting points will

be *shared* among all penalization levels. Note that if a the starting point is *specific* to a penalization level, this penalization level is added to the grid of penalization levels (either the manually specified grid in `lambda` or the automatically generated grid of size `nlambda`). If `standardize = TRUE`, the starting points are also scaled.

`intercept` include an intercept in the model.

`bdp` desired breakdown point of the estimator, between 0.05 and 0.5. The actual breakdown point may be slightly larger/smaller to avoid instabilities of the S-loss.

`cc` tuning constant for the S-estimator. Default is chosen based on the breakdown point `bdp`. This affects the estimated coefficients only if `standardize=TRUE`. Otherwise only the estimated scale of the residuals would be affected.

`eps` numerical tolerance.

`explore_solutions` number of solutions to keep after the exploration step. The best `explore_solutions` are then iterated to full numerical tolerance `eps`. If 0, all non-duplicated solutions are kept.

`explore_tol,explore_it` numerical tolerance and maximum number of iterations for exploring possible solutions. The tolerance should be (much) looser than `eps` to be useful, and the number of iterations should also be much smaller than the maximum number of iterations given via `algorithm_opts`. `explore_tol` is also used to determine if two solutions are equal in the exploration stage.

`max_solutions` retain only up to `max_solutions` unique solutions per penalization level.

`comparison_tol` numeric tolerance to determine if two solutions are equal. The comparison is first done on the absolute difference in the value of the objective function at the solution. If this is less than `comparison_tol`, two solutions are deemed equal if the squared difference of the intercepts is less than `comparison_tol` and the squared L_2 norm of the difference vector is less than `comparison_tol`.

`add_zero_based` also consider the 0-based regularization path. See details for a description.

`enpy_specific` use the EN-PY initial estimates only at the penalization level they are computed for. See details for a description.

`carry_forward` carry the best solutions forward to the next penalty level.

`sparse` use sparse coefficient vectors.

`ncores` number of CPU cores to use in parallel. By default, only one CPU core is used. Not supported on all platforms, in which case a warning is given.

`algorithm_opts` options for the MM algorithm to compute the estimates. See `mm_algorithm_options()` for details.

`mscale_opts` options for the M-scale estimation. See `mscale_algorithm_options()` for details.

`enpy_opts` options for the ENPY initial estimates, created with the `enpy_options()` function. See `enpy_initial_estimates()` for details.

`alpha` elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. `alpha = 1` is the LASSO penalty, and `alpha = 0` the Ridge penalty. Can be a vector of several values, but `alpha = 0` cannot be mixed with other values.

`alpha_preliminary`
: `alpha` parameter for the preliminary estimate.

`exponent`
: the exponent for computing the penalty loadings based on the preliminary estimate.

Details

The built-in CV metrics are

`"tau_size"` τ-size of the prediction error, computed by `tau_size()` (default).

`"mape"` Median absolute prediction error.

`"rmspe"` Root mean squared prediction error.

`"auroc"` Area under the receiver operator characteristic curve (actually 1 - AUROC). Only sensible for binary responses.

`adapense_cv()` is a convenience wrapper which performs 3 steps:

1. compute preliminary estimates via `pense_cv(..., alpha = alpha_preliminary)`,
2. computes the penalty loadings from the estimate `beta` with best prediction performance by `adapense_loadings = 1 / abs(beta)^exponent`, and
3. compute the adaptive PENSE estimates via `pense_cv(..., penalty_loadings = adapense_loadings)`.

Value

a list-like object with the same components as returned by `pense()`, plus the following:

`cvres` data frame of average cross-validated performance.

a list-like object as returned by `pense_cv()` plus the following

`preliminary` the CV results for the preliminary estimate.

`exponent` exponent used to compute the penalty loadings.

`penalty_loadings` penalty loadings used for the adaptive PENSE estimate.

See Also

`pense()` for computing regularized S-estimates without cross-validation.

`coef.pense_cvfit()` for extracting coefficient estimates.

`plot.pense_cvfit()` for plotting the CV performance or the regularization path.

Other functions to compute robust estimates with CV: `change_cv_measure()`, `regmest_cv()`

Other functions to compute robust estimates with CV: `change_cv_measure()`, `regmest_cv()`

Examples

```
# Compute the adaptive PENSE regularization path for Freeny's
# revenue data (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

## Either use the convenience function directly ...
set.seed(123)
ada_convenience <- adapense_cv(x, freeny$y, alpha = 0.5,
                               cv_repl = 2, cv_k = 4)

## ... or compute the steps manually:
# Step 1: Compute preliminary estimates with CV
set.seed(123)
preliminary_estimate <- pense_cv(x, freeny$y, alpha = 0,
                                 cv_repl = 2, cv_k = 4)
plot(preliminary_estimate, se_mult = 1)

# Step 2: Use the coefficients with best prediction performance
# to define the penalty loadings:
prelim_coefs <- coef(preliminary_estimate, lambda = 'min')
pen_loadings <- 1 / abs(prelim_coefs[-1])

# Step 3: Compute the adaptive PENSE estimates and estimate
# their prediction performance.
set.seed(123)
ada_manual <- pense_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 2, cv_k = 4,
                       penalty_loadings = pen_loadings)

# Visualize the prediction performance and coefficient path of
# the adaptive PENSE estimates (manual vs. automatic)
def.par <- par(no.readonly = TRUE)
layout(matrix(1:4, ncol = 2, byrow = TRUE))
plot(ada_convenience$preliminary)
plot(preliminary_estimate)
plot(ada_convenience)
plot(ada_manual)
par(def.par)
```

`plot.pense_cvfit` *Plot Method for Penalized Estimates With Cross-Validation*

Description

Plot the cross-validation performance or the coefficient path for fitted penalized elastic net S- or LS-estimates of regression.

Usage

```
## S3 method for class 'pense_cvfit'
plot(x, what = c("cv", "coef.path"), alpha = NULL, se_mult = 1, ...)
```

Arguments

x	fitted estimates with cross-validation information.
what	plot either the CV performance or the coefficient path.
alpha	If what = "cv", only CV performance for fits with matching alpha are plotted. In case alpha is missing or NULL, all fits in x are plotted. If what = "coef.path", plot the coefficient path for the fit with the given hyper-parameter value or, in case alpha is missing, for the first value in x$alpha.
se_mult	if plotting CV performance, multiplier of the estimated SE.
...	currently ignored.

See Also

Other functions for plotting and printing: plot.pense_fit(), prediction_performance(), summary.pense_cvfit()

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- pense(x, freeny$y, alpha = 0.5)
plot(regpath)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- pense_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

plot.pense_fit *Plot Method for Penalized Estimates*

Description

Plot the coefficient path for fitted penalized elastic net S- or LS-estimates of regression.

Usage

```
## S3 method for class 'pense_fit'
plot(x, alpha, ...)
```

Arguments

x	fitted estimates.
alpha	Plot the coefficient path for the fit with the given hyper-parameter value. If missing of NULL, the first value in x$alpha is used.
...	currently ignored.

See Also

Other functions for plotting and printing: plot.pense_cvfit(), prediction_performance(), summary.pense_cvfit()

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- pense(x, freeny$y, alpha = 0.5)
plot(regpath)

# Extract the coefficients at a certain penalization level
coef(regpath, lambda = regpath$lambda[[1]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- pense_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
coef(cv_results)
```

```
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

`predict.pense_cvfit` *Predict Method for PENSE Fits*

Description

Predict response values using a PENSE (or LS-EN) regularization path with hyper-parameters chosen by cross-validation.

Usage

```
## S3 method for class 'pense_cvfit'
predict(object, newdata, alpha = NULL, lambda = "min", se_mult = 1, ...)
```

Arguments

`object`	PENSE with cross-validated hyper-parameters to extract coefficients from.
`newdata`	an optional matrix of new predictor values. If missing, the fitted values are computed.
`alpha`	Either a single number or `NULL` (default). If given, only fits with the given `alpha` value are considered. If `lambda` is a numeric value and `object` was fit with multiple *alpha* values and no value is provided, the first value in `object$alpha` is used with a warning.
`lambda`	either a string specifying which penalty level to use (`"min"`, `"se"`, `"{m}-se"`) or a single numeric value of the penalty parameter. See details.
`se_mult`	If `lambda = "se"`, the multiple of standard errors to tolerate.
`...`	currently not used.

Value

a numeric vector of residuals for the given penalization level.

Hyper-parameters

If `lambda = "{m}-se"` and `object` contains fitted estimates for every penalization level in the sequence, use the fit the most parsimonious model with prediction performance statistically indistinguishable from the best model. This is determined to be the model with prediction performance within `m * cv_se` from the best model. If `lambda = "se"`, the multiplier *m* is taken from `se_mult`.

By default all *alpha* hyper-parameters available in the fitted object are considered. This can be overridden by supplying one or multiple values in parameter `alpha`. For example, if `lambda = "1-se"` and `alpha` contains two values, the "1-SE" rule is applied individually for each `alpha` value, and the fit with the better prediction error is considered.

In case `lambda` is a number and `object` was fit for several *alpha* hyper-parameters, `alpha` must also be given, or the first value in `object$alpha` is used with a warning.

See Also

Other functions for extracting components: `coef.pense_cvfit()`, `coef.pense_fit()`, `predict.pense_fit()`, `residuals.pense_cvfit()`, `residuals.pense_fit()`

Examples

```
# Compute the LS-EN regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- elnet(x, freeny$y, alpha = 0.75)

# Predict the response using a specific penalization level
predict(regpath, newdata = freeny[1:5, 2:5],
        lambda = regpath$lambda[[1]][[10]])

# Extract the residuals at a certain penalization level
residuals(regpath, lambda = regpath$lambda[[1]][[5]])

# Select penalization level via cross-validation
set.seed(123)
cv_results <- elnet_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 10, cv_k = 4)

# Predict the response using the "best" penalization level
predict(cv_results, newdata = freeny[1:5, 2:5])

# Extract the residuals at the "best" penalization level
residuals(cv_results)
# Extract the residuals at a more parsimonious penalization level
residuals(cv_results, lambda = "1.5-se")
```

---

predict.pense_fit *Predict Method for PENSE Fits*

Description

Predict response values using a PENSE (or LS-EN) regularization path fitted by `pense()`, `regmest()` or `elnet()`.

Usage

```
## S3 method for class 'pense_fit'
predict(object, newdata, alpha = NULL, lambda, ...)
```

Arguments

object	PENSE regularization path to extract residuals from.
newdata	an optional matrix of new predictor values. If missing, the fitted values are computed.
alpha	Either a single number or NULL (default). If given, only fits with the given alpha value are considered. If object was fit with multiple alpha values, and no value is provided, the first value in object$alpha is used with a warning.
lambda	a single number for the penalty level.
...	currently not used.

Value

a numeric vector of residuals for the given penalization level.

See Also

Other functions for extracting components: coef.pense_cvfit(), coef.pense_fit(), predict.pense_cvfit(), residuals.pense_cvfit(), residuals.pense_fit()

Examples

```
# Compute the LS-EN regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- elnet(x, freeny$y, alpha = 0.75)

# Predict the response using a specific penalization level
predict(regpath, newdata = freeny[1:5, 2:5],
        lambda = regpath$lambda[[1]][[10]])

# Extract the residuals at a certain penalization level
residuals(regpath, lambda = regpath$lambda[[1]][[5]])

# Select penalization level via cross-validation
set.seed(123)
cv_results <- elnet_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 10, cv_k = 4)

# Predict the response using the "best" penalization level
predict(cv_results, newdata = freeny[1:5, 2:5])

# Extract the residuals at the "best" penalization level
residuals(cv_results)
# Extract the residuals at a more parsimonious penalization level
residuals(cv_results, lambda = "1.5-se")
```

`prediction_performance` *Prediction Performance of Adaptive PENSE Fits*

Description

Extract the prediction performance of one or more (adaptive) PENSE fits.

Usage

```
prediction_performance(..., alpha = NULL, lambda = "min", se_mult = 1)

## S3 method for class 'pense_pred_perf'
print(x, ...)
```

Arguments

`...`	one or more (adaptive) PENSE fits with cross-validation information.
`alpha`	Either a numeric vector or `NULL` (default). If given, only fits with the given `alpha` value are considered. If `lambda` is a numeric value and `object` was fit with multiple `alpha` values, the parameter `alpha` must not be missing.
`lambda`	either a string specifying which penalty level to use (`"min"`, `"se"`, `"{x}-se"`) or a single numeric value of the penalty parameter. See details.
`se_mult`	If `lambda = "se"`, the multiple of standard errors to tolerate.
`x`	an object with information on prediction performance created with `prediction_performance()`.

Details

If `lambda = "se"` and the cross-validation was performed with multiple replications, use the penalty level whit prediction performance within `se_mult` of the best prediction performance.

Value

a data frame with details about the prediction performance of the given PENSE fits. The data frame has a custom print method summarizing the prediction performances.

See Also

`summary.pense_cvfit()` for a summary of the fitted model.

Other functions for plotting and printing: `plot.pense_cvfit()`, `plot.pense_fit()`, `summary.pense_cvfit()`

prinsens *Principal Sensitivity Components*

Description

Compute Principal Sensitivity Components for Elastic Net Regression

Usage

```
prinsens(
  x,
  y,
  alpha,
  lambda,
  intercept = TRUE,
  penalty_loadings,
  en_algorithm_opts,
  eps = 1e-06,
  sparse = FALSE,
  ncores = 1L
)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n.
alpha	elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. alpha = 1 is the LASSO penalty, and alpha = 0 the Ridge penalty. Can be a vector of several values, but alpha = 0 cannot be mixed with other values.
lambda	optional user-supplied sequence of penalization levels. If given and not NULL, nlambda and lambda_min_ratio are ignored.
intercept	include an intercept in the model.
penalty_loadings	a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. Only allowed for alpha > 0.
en_algorithm_opts	options for the LS-EN algorithm. See en_algorithm_options for details.
eps	numerical tolerance.
sparse	use sparse coefficient vectors.
ncores	number of CPU cores to use in parallel. By default, only one CPU core is used. Not supported on all platforms, in which case a warning is given.

Value

a list of principal sensitivity components, one per element in lambda. Each PSC is itself a list with items lambda, alpha, and pscs.

References

Cohen Freue, G.V.; Kepplinger, D.; Salibián-Barrera, M.; Smucler, E. Robust elastic net estimators for variable selection and identification of proteomic biomarkers. *Ann. Appl. Stat.* **13** (2019), no. 4, 2065–2090 doi:10.1214/19AOAS1269

Pena, D., and Yohai, V.J. A Fast Procedure for Outlier Diagnostics in Large Regression Problems. *J. Amer. Statist. Assoc.* **94** (1999). no. 446, 434–445. doi:10.2307/2670164

See Also

Other functions for initial estimates: `enpy_initial_estimates()`, `enpy_options()`, `starting_point()`

`regmest` *Compute (Adaptive) Elastic Net M-Estimates of Regression*

Description

Compute elastic net M-estimates along a grid of penalization levels with optional penalty loadings for adaptive elastic net.

Usage

```
regmest(
  x,
  y,
  alpha,
  nlambda = 50,
  lambda,
  lambda_min_ratio,
  scale,
  starting_points,
  penalty_loadings,
  intercept = TRUE,
  eff = 0.9,
  rho = "mopt",
  cc,
  eps = 1e-06,
  explore_solutions = 10,
  explore_tol = 0.1,
  max_solutions = 1,
  comparison_tol = sqrt(eps),
  sparse = FALSE,
  ncores = 1,
  standardize = TRUE,
  algorithm_opts = mm_algorithm_options(),
  add_zero_based = TRUE,
  mscale_bdp = 0.25,
  mscale_opts = mscale_algorithm_options()
)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n. For binary classification, y should be a factor with 2 levels.
alpha	elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. alpha = 1 is the LASSO penalty, and alpha = 0 the Ridge penalty.
nlambda	number of penalization levels.
lambda	optional user-supplied sequence of penalization levels. If given and not NULL, nlambda and lambda_min_ratio are ignored.
lambda_min_ratio	Smallest value of the penalization level as a fraction of the largest level (i.e., the smallest value for which all coefficients are zero). The default depends on the sample size relative to the number of variables and alpha. If more observations than variables are available, the default is 1e-3 * alpha, otherwise 1e-2 * alpha.
scale	fixed scale of the residuals.
starting_points	a list of staring points, created by starting_point(). The starting points are shared among all penalization levels.
penalty_loadings	a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. Only allowed for alpha > 0.
intercept	include an intercept in the model.
eff	the desired asymptotic efficiency of the M-estimator under the Normal model.
rho	which ρ function to use (see rho_function() for the list of supported options).
cc	manually specified cutoff constant for the chosen ρ function. If specified, overrides the eff argument.
eps	numerical tolerance.
explore_solutions	number of solutions to compute up to the desired precision eps.
explore_tol	numerical tolerance for exploring possible solutions. Should be (much) looser than eps to be useful.
max_solutions	only retain up to max_solutions unique solutions per penalization level.
comparison_tol	numeric tolerance to determine if two solutions are equal. The comparison is first done on the absolute difference in the value of the objective function at the solution. If this is less than comparison_tol, two solutions are deemed equal if the squared difference of the intercepts is less than comparison_tol and the squared L_2 norm of the difference vector is less than comparison_tol.
sparse	use sparse coefficient vectors.
ncores	number of CPU cores to use in parallel. By default, only one CPU core is used. Not supported on all platforms, in which case a warning is given.

`standardize` logical flag to standardize the `x` variables prior to fitting the M-estimates. Coefficients are always returned on the original scale. This can fail for variables with a large proportion of a single value (e.g., zero-inflated data). In this case, either compute with `standardize = FALSE` or standardize the data manually.

`algorithm_opts` options for the MM algorithm to compute estimates. See `mm_algorithm_options()` for details.

`add_zero_based` also consider the 0-based regularization path in addition to the given starting points.

`mscale_bdp, mscale_opts` options for the M-scale estimate used to standardize the predictors (if `standardize = TRUE`).

Value

a list-like object with the following items

`alpha` the sequence of `alpha` parameters.

`lambda` a list of sequences of penalization levels, one per `alpha` parameter.

`scale` the used scale of the residuals.

`estimates` a list of estimates. Each estimate contains the following information:

- `intercept` intercept estimate.
- `beta` beta (slope) estimate.
- `lambda` penalization level at which the estimate is computed.
- `alpha` *alpha* hyper-parameter at which the estimate is computed.
- `objf_value` value of the objective function at the solution.
- `statuscode` if `> 0` the algorithm experienced issues when computing the estimate.
- `status` optional status message from the algorithm.

`call` the original call.

See Also

`regmest_cv()` for selecting hyper-parameters via cross-validation.

`coef.pense_fit()` for extracting coefficient estimates.

`plot.pense_fit()` for plotting the regularization path.

Other functions to compute robust estimates: `pense()`

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- regmest(x, freeny$y, alpha = c(0.5, 0.85), scale = 2)
plot(regpath)
```

```
# Extract the coefficients at a certain penalization level
coef(regpath, alpha = 0.85, lambda = regpath$lambda[[2]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- regmest_cv(x, freeny$y, alpha = c(0.5, 0.85), scale = 2,
                         cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
coef(cv_results)
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

`regmest_cv` *Cross-validation for (Adaptive) Elastic Net M-Estimates*

Description

Perform (repeated) K-fold cross-validation for `regmest()`.

`adamest_cv()` is a convenience wrapper to compute adaptive elastic-net M-estimates.

Usage

```
regmest_cv(
  x,
  y,
  standardize = TRUE,
  lambda,
  cv_k,
  cv_repl = 1,
  cv_type = "naive",
  cv_metric = c("tau_size", "mape", "rmspe", "auroc"),
  fit_all = TRUE,
  cl = NULL,
  ...
)

adamest_cv(x, y, alpha, alpha_preliminary = 0, exponent = 1, ...)
```

Arguments

x	n by p matrix of numeric predictors.
y	vector of response values of length n. For binary classification, y should be a factor with 2 levels.
standardize	whether to standardize the x variables prior to fitting the PENSE estimates. Can also be set to "cv_only", in which case the input data is not standardized, but the training data in the CV folds is scaled to match the scaling of the input data. Coefficients are always returned on the original scale. This can fail for variables with a large proportion of a single value (e.g., zero-inflated data). In this case, either compute with standardize = FALSE or standardize the data manually.
lambda	optional user-supplied sequence of penalization levels. If given and not NULL, nlambda and lambda_min_ratio are ignored.
cv_k	number of folds per cross-validation.
cv_repl	number of cross-validation replications.
cv_type	what kind of cross-validation should be performed: robust information sharing (ris) or standard (naive) CV.
cv_metric	only for cv_type='naive'. Either a string specifying the performance metric to use, or a function to evaluate prediction errors in a single CV replication. If a function, the number of arguments define the data the function receives. If the function takes a single argument, it is called with a single numeric vector of prediction errors. If the function takes two or more arguments, it is called with the predicted values as first argument and the true values as second argument. The function must always return a single numeric value quantifying the prediction performance. The order of the given values corresponds to the order in the input data.
fit_all	only for cv_type='naive'. If TRUE, fit the model for all penalization levels. Can also be any combination of "min" and "{x}-se", in which case only models at the penalization level with smallest average CV accuracy, or within {x} standard errors, respectively. Setting fit_all to FALSE is equivalent to "min". Applies to all alpha value.
cl	a parallel cluster. Can only be used in combination with ncores = 1.
...	Arguments passed on to regmest scale fixed scale of the residuals. nlambda number of penalization levels. lambda_min_ratio Smallest value of the penalization level as a fraction of the largest level (i.e., the smallest value for which all coefficients are zero). The default depends on the sample size relative to the number of variables and alpha. If more observations than variables are available, the default is 1e-3 * alpha, otherwise 1e-2 * alpha. penalty_loadings a vector of positive penalty loadings (a.k.a. weights) for different penalization of each coefficient. Only allowed for alpha > 0. starting_points a list of staring points, created by starting_point(). The starting points are shared among all penalization levels. intercept include an intercept in the model.

`add_zero_based` also consider the 0-based regularization path in addition to the given starting points.

`rho` which ρ function to use (see `rho_function()` for the list of supported options).

`eff` the desired asymptotic efficiency of the M-estimator under the Normal model.

`cc` manually specified cutoff constant for the chosen ρ function. If specified, overrides the `eff` argument.

`eps` numerical tolerance.

`explore_solutions` number of solutions to compute up to the desired precision `eps`.

`explore_tol` numerical tolerance for exploring possible solutions. Should be (much) looser than `eps` to be useful.

`max_solutions` only retain up to `max_solutions` unique solutions per penalization level.

`comparison_tol` numeric tolerance to determine if two solutions are equal. The comparison is first done on the absolute difference in the value of the objective function at the solution. If this is less than `comparison_tol`, two solutions are deemed equal if the squared difference of the intercepts is less than `comparison_tol` and the squared L_2 norm of the difference vector is less than `comparison_tol`.

`sparse` use sparse coefficient vectors.

`ncores` number of CPU cores to use in parallel. By default, only one CPU core is used. Not supported on all platforms, in which case a warning is given.

`algorithm_opts` options for the MM algorithm to compute estimates. See `mm_algorithm_options()` for details.

`mscale_bdp,mscale_opts` options for the M-scale estimate used to standardize the predictors (if `standardize = TRUE`).

`alpha` elastic net penalty mixing parameter with $0 \leq \alpha \leq 1$. `alpha = 1` is the LASSO penalty, and `alpha = 0` the Ridge penalty.

`alpha_preliminary`
`alpha` parameter for the preliminary estimate.

`exponent` the exponent for computing the penalty loadings based on the preliminary estimate.

Details

The built-in CV metrics are

`"tau_size"` τ-size of the prediction error, computed by `tau_size()` (default).

`"mape"` Median absolute prediction error.

`"rmspe"` Root mean squared prediction error.

`"auroc"` Area under the receiver operator characteristic curve (actually 1 - AUROC). Only sensible for binary responses.

`adamest_cv()` is a convenience wrapper which performs 3 steps:

1. compute preliminary estimates via `regmest_cv(..., alpha = alpha_preliminary)`,
2. computes the penalty loadings from the estimate `beta` with best prediction performance by `adamest_loadings = 1 / abs(beta)^exponent`, and
3. compute the adaptive PENSE estimates via `regmest_cv(..., penalty_loadings = adamest_loadings)`.

Value

a list-like object as returned by `regmest()`, plus the following components:

`cvres` data frame of average cross-validated performance.

a list-like object as returned by `adamest_cv()` plus the following components:

`exponent` value of the exponent.

`preliminary` CV results for the preliminary estimate.

`penalty_loadings` penalty loadings used for the adaptive elastic net M-estimate.

See Also

`regmest()` for computing regularized S-estimates without cross-validation.

`coef.pense_cvfit()` for extracting coefficient estimates.

`plot.pense_cvfit()` for plotting the CV performance or the regularization path.

Other functions to compute robust estimates with CV: `change_cv_measure()`, `pense_cv()`

Other functions to compute robust estimates with CV: `change_cv_measure()`, `pense_cv()`

Examples

```
# Compute the PENSE regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- regmest(x, freeny$y, alpha = c(0.5, 0.85), scale = 2)
plot(regpath)

# Extract the coefficients at a certain penalization level
coef(regpath, alpha = 0.85, lambda = regpath$lambda[[2]][[40]])

# What penalization level leads to good prediction performance?
set.seed(123)
cv_results <- regmest_cv(x, freeny$y, alpha = c(0.5, 0.85), scale = 2,
                         cv_repl = 2, cv_k = 4)
plot(cv_results, se_mult = 1)

# Print a summary of the fit and the cross-validation results.
summary(cv_results)

# Extract the coefficients at the penalization level with
# smallest prediction error ...
coef(cv_results)
```

```
# ... or at the penalization level with prediction error
# statistically indistinguishable from the minimum.
coef(cv_results, lambda = '1-se')
```

`residuals.pense_cvfit` *Extract Residuals*

Description

Extract residuals from a PENSE (or LS-EN) regularization path with hyper-parameters chosen by cross-validation.

Usage

```
## S3 method for class 'pense_cvfit'
residuals(object, alpha = NULL, lambda = "min", se_mult = 1, ...)
```

Arguments

`object`	PENSE with cross-validated hyper-parameters to extract coefficients from.
`alpha`	Either a single number or `NULL` (default). If given, only fits with the given `alpha` value are considered. If `lambda` is a numeric value and `object` was fit with multiple *alpha* values and no value is provided, the first value in `object$alpha` is used with a warning.
`lambda`	either a string specifying which penalty level to use (`"min"`, `"se"`, `"{m}-se"`) or a single numeric value of the penalty parameter. See details.
`se_mult`	If `lambda = "se"`, the multiple of standard errors to tolerate.
`...`	currently not used.

Value

a numeric vector of residuals for the given penalization level.

Hyper-parameters

If `lambda = "{m}-se"` and `object` contains fitted estimates for every penalization level in the sequence, use the fit the most parsimonious model with prediction performance statistically indistinguishable from the best model. This is determined to be the model with prediction performance within `m * cv_se` from the best model. If `lambda = "se"`, the multiplier *m* is taken from `se_mult`.

By default all *alpha* hyper-parameters available in the fitted object are considered. This can be overridden by supplying one or multiple values in parameter `alpha`. For example, if `lambda = "1-se"` and `alpha` contains two values, the "1-SE" rule is applied individually for each `alpha` value, and the fit with the better prediction error is considered.

In case `lambda` is a number and `object` was fit for several *alpha* hyper-parameters, `alpha` must also be given, or the first value in `object$alpha` is used with a warning.

See Also

Other functions for extracting components: coef.pense_cvfit(), coef.pense_fit(), predict.pense_cvfit(), predict.pense_fit(), residuals.pense_fit()

Examples

```
# Compute the LS-EN regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- elnet(x, freeny$y, alpha = 0.75)

# Predict the response using a specific penalization level
predict(regpath, newdata = freeny[1:5, 2:5],
        lambda = regpath$lambda[[1]][[10]])

# Extract the residuals at a certain penalization level
residuals(regpath, lambda = regpath$lambda[[1]][[5]])

# Select penalization level via cross-validation
set.seed(123)
cv_results <- elnet_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 10, cv_k = 4)

# Predict the response using the "best" penalization level
predict(cv_results, newdata = freeny[1:5, 2:5])

# Extract the residuals at the "best" penalization level
residuals(cv_results)
# Extract the residuals at a more parsimonious penalization level
residuals(cv_results, lambda = "1.5-se")
```

---

residuals.pense_fit *Extract Residuals*

Description

Extract residuals from a PENSE (or LS-EN) regularization path fitted by pense(), regmest() or elnet().

Usage

```
## S3 method for class 'pense_fit'
residuals(object, alpha = NULL, lambda, ...)
```

Arguments

object	PENSE regularization path to extract residuals from.
alpha	Either a single number or NULL (default). If given, only fits with the given alpha value are considered. If object was fit with multiple alpha values, and no value is provided, the first value in object$alpha is used with a warning.
lambda	a single number for the penalty level.
...	currently not used.

Value

a numeric vector of residuals for the given penalization level.

See Also

Other functions for extracting components: coef.pense_cvfit(), coef.pense_fit(), predict.pense_cvfit(), predict.pense_fit(), residuals.pense_cvfit()

Examples

```
# Compute the LS-EN regularization path for Freeny's revenue data
# (see ?freeny)
data(freeny)
x <- as.matrix(freeny[ , 2:5])

regpath <- elnet(x, freeny$y, alpha = 0.75)

# Predict the response using a specific penalization level
predict(regpath, newdata = freeny[1:5, 2:5],
        lambda = regpath$lambda[[1]][[10]])

# Extract the residuals at a certain penalization level
residuals(regpath, lambda = regpath$lambda[[1]][[5]])

# Select penalization level via cross-validation
set.seed(123)
cv_results <- elnet_cv(x, freeny$y, alpha = 0.5,
                       cv_repl = 10, cv_k = 4)

# Predict the response using the "best" penalization level
predict(cv_results, newdata = freeny[1:5, 2:5])

# Extract the residuals at the "best" penalization level
residuals(cv_results)
# Extract the residuals at a more parsimonious penalization level
residuals(cv_results, lambda = "1.5-se")
```

rho_function *List Available Rho Functions*

Description

List Available Rho Functions

Usage

```
rho_function(rho, convex_ok = TRUE)
```

Arguments

rho	the name of the ρ function to check for existence.
convex_ok	if convex ρ function is acceptable or not.

Value

if rho is missing returns a vector of supported ρ function names, otherwise the internal integer representation of the ρ function.

See Also

Other Robustness control options: consistency_const(), mscale_algorithm_options()

starting_point *Create Starting Points for the PENSE Algorithm*

Description

Create a starting point for starting the PENSE algorithm in pense(). Multiple starting points can be created by combining starting points via c(starting_point_1, starting_point_2, ...).

Usage

```
starting_point(beta, intercept, lambda, alpha)

as_starting_point(object, specific = FALSE, ...)

## S3 method for class 'enpy_starting_points'
as_starting_point(object, specific = FALSE, ...)

## S3 method for class 'pense_fit'
as_starting_point(object, specific = FALSE, alpha, lambda, ...)

## S3 method for class 'pense_cvfit'
```

```
as_starting_point(
  object,
  specific = FALSE,
  alpha,
  lambda = c("min", "se"),
  se_mult = 1,
  ...
)
```

Arguments

beta	beta coefficients at the starting point. Can be a numeric vector, a sparse vector of class dsparseVector, or a sparse matrix of class dgCMatrix with a single column.
intercept	intercept coefficient at the starting point.
lambda	optionally either a string specifying which penalty level to use ("min" or "se") or a numeric vector of the penalty levels to extract from object. Penalization levels not present in object are ignored with a warning. If NULL, all estimates in object are extracted. If a numeric vector, alpha must be given and a single number.
alpha	optional value for the alpha hyper-parameter. If given, only estimates with matching alpha values are extracted. Values not present in object are ignored with a warning.
object	an object with estimates to use as starting points.
specific	whether the estimates should be used as starting points only at the penalization level they are computed for. Defaults to using the estimates as starting points for all penalization levels.
...	further arguments passed to or from other methods.
se_mult	If lambda = "se", the multiple of standard errors to tolerate.

Details

A starting points can either be *shared*, i.e., used for every penalization level PENSE estimates are computed for, or *specific* to one penalization level. To create a specific starting point, provide the penalization parameters lambda and alpha. If lambda or alpha are missing, a shared starting point is created. Shared and specific starting points can all be combined into a single list of starting points, with pense() handling them correctly. Note that specific starting points will lead to the lambda value being added to the grid of penalization levels. See pense() for details.

Starting points computed via enpy_initial_estimates() are by default *shared* starting points but can be transformed to *specific* starting points via as_starting_point(..., specific = TRUE).

When creating starting points from cross-validated fits, it is possible to extract only the estimate with best CV performance (lambda = "min"), or the estimate with CV performance statistically indistinguishable from the best performance (lambda = "se"). This is determined to be the estimate with prediction performance within se_mult * cv_se from the best model.

Value

an object of type starting_points to be used as starting point for pense().

See Also

Other functions for initial estimates: `enpy_initial_estimates()`, `enpy_options()`, `prinsens()`

`summary.pense_cvfit` *Summarize Cross-Validated PENSE Fit*

Description

If `lambda = "se"` and `object` contains fitted estimates for every penalization level in the sequence, extract the coefficients of the most parsimonious model with prediction performance statistically indistinguishable from the best model. This is determined to be the model with prediction performance within `se_mult * cv_se` from the best model.

Usage

```
## S3 method for class 'pense_cvfit'
summary(object, alpha, lambda = "min", se_mult = 1, ...)

## S3 method for class 'pense_cvfit'
print(x, alpha, lambda = "min", se_mult = 1, ...)
```

Arguments

`object`, `x`	an (adaptive) PENSE fit with cross-validation information.
`alpha`	Either a single number or missing. If given, only fits with the given `alpha` value are considered. If `lambda` is a numeric value and `object` was fit with multiple `alpha` values, the parameter `alpha` must not be missing.
`lambda`	either a string specifying which penalty level to use (`"min"`, `"se"`, `"{x}-se"`) or a single numeric value of the penalty parameter. See details.
`se_mult`	If `lambda = "se"`, the multiple of standard errors to tolerate.
`...`	ignored.

See Also

`prediction_performance()` for information about the estimated prediction performance.

`coef.pense_cvfit()` for extracting only the estimated coefficients.

Other functions for plotting and printing: `plot.pense_cvfit()`, `plot.pense_fit()`, `prediction_performance()`

tau_size *Compute the Tau-Scale of Centered Values*

Description

Compute the τ-scale without centering the values.

Usage

```
tau_size(x)
```

Arguments

x	numeric values. Missing values are verbosely ignored.

Value

the τ estimate of scale of centered values.

See Also

Other functions to compute robust estimates of location and scale: `mloc()`, `mlocscale()`, `mscale()`

Index

∗ **LS-EN algorithm options**
en_admm_options, 14
en_algorithm_options, 15
en_cd_options, 15
en_dal_options, 16
en_lars_options, 17
∗ **Robust EN algorithms**
cd_algorithm_options, 3
mm_algorithm_options, 22
∗ **Robustness control options**
consistency_const, 8
mscale_algorithm_options, 23
rho_function, 50
∗ **functions for computing non-robust estimates**
elnet, 9
elnet_cv, 11
∗ **functions for extracting components**
coef.pense_cvfit, 4
coef.pense_fit, 6
predict.pense_cvfit, 35
predict.pense_fit, 36
residuals.pense_cvfit, 47
residuals.pense_fit, 48
∗ **functions for initial estimates**
enpy_initial_estimates, 17
enpy_options, 19
prinsens, 39
starting_point, 50
∗ **functions for plotting and printing**
plot.pense_cvfit, 32
plot.pense_fit, 34
prediction_performance, 38
summary.pense_cvfit, 52
∗ **functions to compute robust estimates of location and scale**
mloc, 20
mlocscale, 21
mscale, 22
tau_size, 53
∗ **functions to compute robust estimates with CV**
change_cv_measure, 4
pense_cv, 28
regmest_cv, 43
∗ **functions to compute robust estimates**
pense, 24
regmest, 40

adaelnet *(*elnet*)*, 9
adaen *(*elnet*)*, 9
adamest_cv *(*regmest_cv*)*, 43
adamest_cv(), *46*
adapense *(*pense*)*, 24
adapense_cv *(*pense_cv*)*, 28
as_starting_point *(*starting_point*)*, 50
as_starting_point(), *18*

cd_algorithm_options, 3, *22*
change_cv_measure, 4, *31*, *46*
coef.pense_cvfit, 4, *7*, *36, 37*, *48, 49*
coef.pense_cvfit(), *7*, *13*, *31*, *46*, *52*
coef.pense_fit, 6, *6*, *36, 37*, *48, 49*
coef.pense_fit(), *10*, *27*, *42*
consistency_const, 8, *23*, *50*

dgCMatrix, *51*
dsparseVector, *5*, *7*, *51*

efficiency_const *(*consistency_const*)*, 8
elnet, 9, *12, 13*
elnet(), *6*, *11*, *13*, *36*, *48*
elnet_cv, *10*, 11
en_admm_options, 14, *15–17*
en_admm_options(), *15*
en_algorithm_options, *10*, *13, 14*, 15, *16, 17*, *19*, *22*, *39*
en_cd_options, *14*, 15, *15–17*
en_cd_options(), *15*

en_dal_options, *14, 15*, 16, *16, 17*
en_dal_options(), *15*
en_lars_options, *14–16*, 17
en_lars_options(), *15*
enpy_initial_estimates, 17, *20*, *40*, *52*
enpy_initial_estimates(), *25, 26*, *29, 30*, *51*
enpy_options, *18*, 19, *40*, *52*
enpy_options(), *18*, *26*, *30*

mad(), *20*
mloc, 20, *21*, *23*, *53*
mlocscale, *20*, 21, *23*, *53*
mm_algorithm_options, *3*, 22
mm_algorithm_options(), *26*, *30*, *42*, *45*
mscale, *20, 21*, 22, *53*
mscale_algorithm_options, *8*, 23, *50*
mscale_algorithm_options(), *18*, *21*, *23*, *26*, *30*

parallel, *12*, *29*, *44*
pense, 24, *29*, *42*
pense(), *6*, *10*, *18*, *28*, *31*, *36*, *48*, *50, 51*
pense_cv, *4*, 28, *46*
pense_cv(), *13*, *27*, *31*
plot.pense_cvfit, 32, *34*, *38*, *52*
plot.pense_cvfit(), *13*, *31*, *46*
plot.pense_fit, *33*, 34, *38*, *52*
plot.pense_fit(), *10*, *27*, *42*
predict.pense_cvfit, *6, 7*, 35, *37*, *48, 49*
predict.pense_fit, *6, 7*, 36, *36*, *48, 49*
prediction_performance, *33, 34*, 38, *52*
prediction_performance(), *52*
prinsens, *18*, *20*, 39, *52*
print.pense_cvfit *(*summary.pense_cvfit*)*, 52
print.pense_pred_perf *(*prediction_performance*)*, 38

regmest, *27*, 40, *44*
regmest(), *36*, *43*, *46*, *48*
regmest_cv, *4*, *31*, 43
regmest_cv(), *42*
residuals.pense_cvfit, *6, 7*, *36, 37*, 47, *49*
residuals.pense_fit, *6, 7*, *36, 37*, 48, *48*
rho_function, *8*, *23*, 50
rho_function(), *8*, *20, 21*, *23*, *41*, *45*

starting_point, *18*, *20*, *40*, 50
starting_point(), *25*, *29*, *41*, *44*
summary.pense_cvfit, *33, 34*, *38*, 52
summary.pense_cvfit(), *38*

tau_size, *20, 21*, *23*, 53
tau_size(), *13*, *31*, *45*