Package 'psvr'

September 24, 2026

Title Percentage-Error Support Vector Regression

Version 0.1.0

Description Implements four support vector regression (SVR) models derived from a unified mathematical framework for percentage-error loss functions: epsilon-SVR minimizing the mean absolute percentage error (MAPE), its symmetric kernel extension, least-squares SVR (LS-SVR) minimizing the root mean square percentage error (RMSPE), and its symmetric counterpart. All models require strictly positive targets. The epsilon-SVR models are solved via a built-in sequential minimal optimization (SMO) algorithm (with 'osqp' available as an optional alternative backend) and the LS-SVR models via a linear system (base R). See Benavides-Herrera et al. (2026) <doi:10.3390/math14101679> for the mathematical derivations.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1)

Imports dials, Matrix, parsnip, Rcpp (>= 1.0.10), rlang, scales, stats, tune, utils, workflowsets

LinkingTo Rcpp

Suggests dplyr, ggplot2, knitr, osqp, recipes, rmarkdown, rsample, testthat (>= 3.0.0), tibble, workflows, yardstick

Config/testthat/edition 3

VignetteBuilder knitr

URL https://pbenavidesh.github.io/psvr/

Config/Needs/website rmarkdown, tsibbledata, tsibble, lubridate, ggplot2, kableExtra, kernlab, scales, tidyr, tidyverse, tidymodels, modeldata, future, ranger, xgboost, ggrepel, sessioninfo, modeltime, timetk, randomForest, readxl, xfun, e1071, quantreg, lars, patchwork, here

NeedsCompilation yes

Author Pablo Benavides-Herrera [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4926-4763>)

Maintainer Pablo Benavides-Herrera <pbenavides@iteso.mx>

Repository CRAN

Date/Publication 2026-09-24 14:40:08 UTC

Contents

`coef.psvr_mape` *Extract coefficients from a psvr_mape model*

Description

Extract coefficients from a psvr_mape model

Usage

```
## S3 method for class 'psvr_mape'
coef(object, ...)
```

Arguments

`object`	An object of class `"psvr_mape"`.
`...`	Ignored.

Value

A named list with five components:

`alpha`, `alpha_star` The length-`N` pre-pruning dual variables α_k and α_k^*.

`beta` The pruned dual differences $\beta_k = \alpha_k - \alpha_k^*$ over the support-vector indices (length `n_sv`); this is what `predict()` uses.

`b` Bias term.

`support_data` Support vector input matrix.

The LS-SVR classes return **three** components rather than five, since they have no `alpha_star` and no pruned `beta`; the absent components are not materialised as `NULL`. So `names(coef(fit))` depends on the model family. That is deliberate: each class is family-specific, and inventing empty slots to make the two agree would add structure with nothing to inherit it from.

Renamed in 0.0.2.9011

`alpha` previously held the pruned β and `support_data` was named `X_sv`, which made `coef(fit)$alpha` mean the length-`n_sv` β here but the length-`N` dual α on a fit from the superseded `psvr()`: one generic returning two different vectors under one name, silently, depending on entry point. The β-under-`alpha` meaning is the one 0.0.2.9004 moved away from on the object itself; this aligns `coef()` with it.

coef.psvr_mape_sym *Extract coefficients from a psvr_mape_sym model*

Description

Extract coefficients from a psvr_mape_sym model

Usage

```
## S3 method for class 'psvr_mape_sym'
coef(object, ...)
```

Arguments

object	An object of class "psvr_mape_sym".
...	Ignored.

Value

A named list with five components, identical in meaning to coef.psvr_mape():

alpha, alpha_star The length-N pre-pruning dual variables α_k and α_k^*.

beta The pruned dual differences $\beta_k = \alpha_k - \alpha_k^*$ over the support-vector indices (length n_sv); this is what predict() uses.

b Bias term.

support_data Support vector input matrix.

Renamed in 0.0.2.9011

See coef.psvr_mape() — the same rename, for the same reason, applied to both MAPE classes together.

coef.psvr_rmspe *Extract coefficients from a psvr_rmspe model*

Description

Extract coefficients from a psvr_rmspe model

Usage

```
## S3 method for class 'psvr_rmspe'
coef(object, ...)
```

Arguments

object	An object of class "psvr_rmspe".
...	Ignored.

Value

A named list with components:

alpha Dual variables / Lagrange multipliers (length N).

b Bias term.

support_data Training input matrix (all N observations).

Three components, against five for the MAPE classes (coef.psvr_mape()): LS-SVR has no alpha_star and no pruned beta, and they are **not** materialised as NULL. So names(coef(fit)) depends on the model family, which is a decision rather than an oversight – each class is family-specific, and inventing empty slots to make the two agree would add structure with nothing to inherit it from. $alpha_star and $beta yield NULL on both, so every accessor still agrees.

Renamed in 0.0.2.9011

support_data was named X_sv. LS-SVR performs no pruning — every training point contributes to f(x) — so there are no support vectors to name: it was an epsilon-SVR name on an LS-SVR value. The value is unchanged.

---

coef.psvr_rmspe_sym *Extract coefficients from a psvr_rmspe_sym model*

Description

Extract coefficients from a psvr_rmspe_sym model

Usage

```
## S3 method for class 'psvr_rmspe_sym'
coef(object, ...)
```

Arguments

object	An object of class "psvr_rmspe_sym".
...	Ignored.

Value

A named list with components:

`alpha` Dual variables / Lagrange multipliers (length N).

`b` Bias term.

`support_data` Training input matrix (all N observations).

Three components, not the five the MAPE classes return: LS-SVR has no `alpha_star` and no pruned `beta`, and they are not materialised as `NULL`. See `coef.psvr_rmspe()`.

Renamed in 0.0.2.9011

See `coef.psvr_rmspe()` — the same rename, for the same reason, applied to both LS-SVR classes together.

`cost_psvr` *Cost parameter with extended range for psvr models*

Description

A dials parameter for the regularisation parameter in psvr models. The default range [-2, 10] on the log2 scale (corresponding to approximately 0.25 to 1024) is wider than `dials::cost()` to accommodate the larger regularisation values typically needed by LS-SVR models.

Usage

```
cost_psvr(range = c(-2, 10), trans = scales::log2_trans())
```

Arguments

`range`	Numeric vector of length 2 on the log2 scale. Default `c(-2, 10)`.
`trans`	A `scales` transformation object. Default `scales::log2_trans()`.

Details

For LS-SVR (`m3`, `m4`) models the static range is still frequently too narrow, because Γ enters the LS-SVR system through the y_k^2/Γ diagonal and so scales with `var(y) * N` rather than sitting at a fixed magnitude. Prefer `cost_psvr_ls_data()`, whose range is computed from the training outcome.

Value

A `quant_param` dials object.

See Also

`cost_psvr_ls_data()` for a data-driven LS-SVR variant.

Examples

```
cost_psvr()
```

`cost_psvr_ls_data` *Data-driven cost range for LS-SVR psvr models*

Description

Returns a `quant_param` whose search range scales with `var(y) * N`, the standard heuristic for the LS-SVR regularisation parameter Γ (Suykens et al. 2002, *Least Squares Support Vector Machines*, §3.1.3). On the log2 scale, the lower bound is `-2` (i.e. $\Gamma \geq 0.25$) and the upper bound is `log2(var(y) * N) + width_log2`.

Usage

```
cost_psvr_ls_data(y, n = length(y), width_log2 = 4)
```

Arguments

`y`	Numeric vector of strictly positive training targets.
`n`	Sample size. Default `length(y)`.
`width_log2`	Scalar giving the half-width (in log2 units) added above `log2(var(y) * n)` to set the upper bound. Default `4` (about 16× headroom). Negative values are accepted with a warning, since the resulting upper bound falls below the `var(y) * n` heuristic and is unlikely to be useful.

Details

The default `width_log2 = 4` places the upper bound about 16 times above `var(y) * n`, which is headroom for a Bayesian or grid search to work in without pinning against the boundary. Because the bound tracks `var(y) * n`, it moves with the outcome: the static `cost_psvr()` range [-2, 10] (i.e. $\Gamma \leq 1024$) is fixed, so it falls short whenever `var(y) * n` exceeds a few hundred — which is the usual case, not the exception.

Use this function for `m3` (LS-SVR) and `m4` (symmetric LS-SVR) workflows. Stick to `cost_psvr()` for `m1`/`m2` (ϵ-SVR), where `cost` maps to `C` and typical optima lie in [10, 100].

Value

A `quant_param` dials object.

See Also

`cost_psvr()`, `psvr_option_add_cost_ls()`

Examples

```
cost_psvr_ls_data(c(10, 20, 30, 40, 50))
```

`make_kernel` *Create a kernel function*

Description

Returns a closure `K(xi, xj)` suitable for use with all four psvr model fitting functions. The returned function accepts numeric vectors of any sign, which is required by the symmetric models (Models 2 and 4) that evaluate `K(xk, -xl)`.

Usage

```
make_kernel(
  type = c("rbf", "linear", "polynomial"),
  sigma = 1,
  degree = 3L,
  coef0 = 1
)
```

Arguments

`type`	Kernel type: `"rbf"`, `"linear"`, or `"polynomial"`.
`sigma`	Bandwidth for the RBF kernel, `sigma > 0` (default `1`).
`degree`	Integer degree for the polynomial kernel, `degree >= 1` (default `3`).
`coef0`	Constant term for the polynomial kernel (default `1`).

Details

The three supported kernels are:

- **RBF:** `K(xi, xj) = exp(-‖xi - xj‖² / (2 * sigma²))`
- **Linear:** `K(xi, xj) = xi · xj`
- **Polynomial:** `K(xi, xj) = (xi · xj + coef0)^degree`

RBF and even-degree polynomial kernels satisfy Assumption 3 of the paper (kernel symmetry), making them compatible with the symmetric models. The linear kernel and odd-degree polynomial kernels do **not** satisfy Assumption 3 and should not be used with the symmetric models, i.e. with `psvr_mape()` / `psvr_rmspe()` or a parsnip spec whose `sym_type` is `"even"` or `"odd"`.

Value

A function `K(xi, xj)` where `xi` and `xj` are numeric vectors of the same length, returning a scalar kernel evaluation.

Examples

```
K <- make_kernel("rbf", sigma = 0.5)
K(c(1, 2), c(3, 4))

K_lin <- make_kernel("linear")
K_lin(c(1, 0), c(0, 1))

K_poly <- make_kernel("polynomial", degree = 2, coef0 = 1)
K_poly(c(1, 2), c(3, 4))
```

---

margin_percentage *Insensitivity margin in percentage units*

Description

A dials parameter for the epsilon tube half-width in psvr MAPE models. Unlike `dials::svm_margin()` which uses absolute units, this parameter is expressed as a percentage of each target value. The default range [1, 20] means the insensitivity tube spans 1% to 20% of each target.

Usage

```
margin_percentage(range = c(1, 20), trans = NULL)
```

Arguments

`range`	Numeric vector of length 2. Default `c(1, 20)`.
`trans`	A `scales` transformation object. Default `NULL`.

Value

A `quant_param` dials object.

Examples

```
margin_percentage()
margin_percentage(range = c(0.5, 10))
```

predict.psvr_mape *Predict from a fitted epsilon-SVR with MAPE model*

Description

Method dispatched on the "psvr_mape" class, which both psvr_mape() and the parsnip engine fit wrappers return.

Usage

```
## S3 method for class 'psvr_mape'
predict(object, newdata, ...)
```

Arguments

object	An object of class "psvr_mape", as returned by psvr_mape() with sym_type = "none" or by the parsnip engine fit wrappers (see psvr-fit-wrappers; sym_type = "even" or "odd" yields "psvr_mape_sym" instead). Unwrap a parsnip fit with parsnip::extract_fit_engine() to obtain it.
newdata	Numeric matrix of new inputs, one observation per row ($M \times p$).
...	Ignored.

Value

Numeric vector of length M with predicted values.

predict.psvr_mape_sym *Predict from a fitted symmetric epsilon-SVR with MAPE model*

Description

Method dispatched on the "psvr_mape_sym" class, which both psvr_mape() and the parsnip engine fit wrappers return. Uses the symmetric representer theorem

$$f(x) = \tfrac{1}{2} \sum_k \beta_k K_s(x_k, x) + b$$

with $K_s(x_k, x) = K(x_k, x) + aK(x_k, -x)$.

Usage

```
## S3 method for class 'psvr_mape_sym'
predict(object, newdata, ...)
```

Arguments

object	An object of class "psvr_mape_sym", as returned by psvr_mape() with sym_type = "even" or "odd", or by the parsnip engine fit wrappers (see psvr-fit-wrappers; sym_type = "none" yields "psvr_mape" instead). Unwrap a parsnip fit with parsnip::extract_fit_engine() to obtain it.
newdata	Numeric matrix of new inputs, one observation per row ($M \times p$).
...	Ignored.

Value

Numeric vector of length M with predicted values.

---

predict.psvr_rmspe *Predict from a fitted LS-SVR with RMSPE model*

Description

Method dispatched on the "psvr_rmspe" class, which both psvr_rmspe() and the parsnip engine fit wrappers return.

Usage

```
## S3 method for class 'psvr_rmspe'
predict(object, newdata, ...)
```

Arguments

object	An object of class "psvr_rmspe", as returned by psvr_rmspe() with sym_type = "none" or by the parsnip engine fit wrappers (see psvr-fit-wrappers; sym_type = "even" or "odd" yields "psvr_rmspe_sym" instead). Unwrap a parsnip fit with parsnip::extract_fit_engine() to obtain it.
newdata	Numeric matrix of new inputs, one observation per row ($M \times p$).
...	Ignored.

Value

Numeric vector of length M with predicted values.

predict.psvr_rmspe_sym *Predict from a fitted symmetric LS-SVR with RMSPE model*

Description

Method dispatched on the "psvr_rmspe_sym" class, which both psvr_rmspe() and the parsnip engine fit wrappers return. Uses the symmetric representer

$$f(x) = \sum_k \alpha_k \cdot \tfrac{1}{2}\left(K(x_k, x) + aK(x_k, -x)\right) + b$$

Usage

```
## S3 method for class 'psvr_rmspe_sym'
predict(object, newdata, ...)
```

Arguments

object	An object of class "psvr_rmspe_sym", as returned by psvr_rmspe() with sym_type = "even" or "odd", or by the parsnip engine fit wrappers (see psvr-fit-wrappers; sym_type = "none" yields "psvr_rmspe" instead). Unwrap a parsnip fit with parsnip::extract_fit_engine() to obtain it.
newdata	Numeric matrix of new inputs, one observation per row ($M \times p$).
...	Ignored.

Value

Numeric vector of length M with predicted values.

print.psvr_mape *Print method for psvr_mape objects*

Description

Print method for psvr_mape objects

Usage

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

Arguments

x	An object of class "psvr_mape".
...	Ignored.

Value

x, invisibly.

---

print.psvr_mape_sym *Print method for psvr_mape_sym objects*

Description

Print method for psvr_mape_sym objects

Usage

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

Arguments

x	An object of class "psvr_mape_sym".
...	Ignored.

Value

x, invisibly.

---

print.psvr_rmspe *Print method for psvr_rmspe objects*

Description

Print method for psvr_rmspe objects

Usage

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

Arguments

x	An object of class "psvr_rmspe".
...	Ignored.

Value

x, invisibly.

print.psvr_rmspe_sym *Print method for psvr_rmspe_sym objects*

Description

Print method for psvr_rmspe_sym objects

Usage

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

Arguments

x	An object of class "psvr_rmspe_sym".
...	Ignored.

Value

x, invisibly.

psvr-fitted *Extract training fitted values from a psvr model*

Description

Returns the length-N in-sample predictions $f(x_k)$ recorded when the model was fitted. No kernel matrix is rebuilt: the values are recovered from state the solver already holds. For the MAPE models that is a matvec against the retained Ω; for the LS-SVR models it is the KKT stationarity identity

$$f(x_k) = y_k - (10^{-6} + y_k^2/\Gamma)\,\alpha_k$$

which costs $O(N)$ and holds in both preconditioner branches. The training inputs X are not retained for this purpose.

Usage

```
## S3 method for class 'psvr_mape'
fitted(object, ...)

## S3 method for class 'psvr_mape_sym'
fitted(object, ...)

## S3 method for class 'psvr_rmspe'
fitted(object, ...)

## S3 method for class 'psvr_rmspe_sym'
fitted(object, ...)
```

Arguments

object	A fitted object of class "psvr_mape", "psvr_mape_sym", "psvr_rmspe" or "psvr_rmspe_sym", from psvr_mape(), psvr_rmspe(), or a parsnip fit unwrapped with parsnip::extract_fit_engine().
...	Ignored.

Details

The result equals predict(object, X_train) to machine precision. It is not bit-identical: the two use different summation orders (a BLAS matvec versus the column-wise reduction in predict()), and for the LS-SVR models the identity above is exact only up to the residual of the linear solve. Observed agreement is within 3e-12 relative across the four models.

Value

Numeric vector of length N (the number of training observations), in training-row order.

Not reachable through parsnip

parsnip registers neither residuals.model_fit nor fitted.model_fit, so calling either generic on a model_fit dispatches to the stats default and returns NULL **silently** - no error, no warning. Reach the psvr object first with parsnip::extract_fit_engine(), then call fitted() or residuals() on that. psvr deliberately does not register S3 methods on parsnip's class. Note also that parsnip::augment() recomputes predictions on whatever new_data it is given and reports response residuals only.

See Also

residuals.psvr_mape() and the other residuals methods

Examples

```
set.seed(1)
X <- matrix(runif(40, 0.5, 3), 20, 2)
y <- 2 + X[, 1]^2
fit <- psvr_rmspe(X, y, kernel = make_kernel("rbf"), gamma = 100)
head(fitted(fit))

# Through parsnip both generics return NULL on the model_fit wrapper;
# extract the engine object first.
df   <- data.frame(x1 = X[, 1], x2 = X[, 2], y = y)
spec <- psvr_rmspe_rbf(cost = 10, rbf_sigma = 0.8)
pfit <- parsnip::fit(spec, y ~ x1 + x2, data = df)
fitted(pfit)                                      # NULL
head(fitted(parsnip::extract_fit_engine(pfit)))  # the fitted values
```

psvr-residuals *Extract training residuals from a psvr model*

Description

Three residual types are available. They are **different quantities with different denominators**, not interchangeable scalings of one another:

Usage

```
## S3 method for class 'psvr_mape'
residuals(object, type = c("response", "percentage", "multiplicative"), ...)

## S3 method for class 'psvr_mape_sym'
residuals(object, type = c("response", "percentage", "multiplicative"), ...)

## S3 method for class 'psvr_rmspe'
residuals(object, type = c("response", "percentage", "multiplicative"), ...)

## S3 method for class 'psvr_rmspe_sym'
residuals(object, type = c("response", "percentage", "multiplicative"), ...)
```

Arguments

`object`	A fitted object of class `"psvr_mape"`, `"psvr_mape_sym"`, `"psvr_rmspe"` or `"psvr_rmspe_sym"`, from `psvr_mape()`, `psvr_rmspe()`, or a parsnip fit unwrapped with `parsnip::extract_fit_engine()`.
`type`	One of `"response"` (default), `"percentage"`, or `"multiplicative"`. See above; the denominators differ.
`...`	Ignored.

Details

`"response"` $y - \hat{y}$. The default, following the R convention for `stats::residuals()`. Units of the response.

`"percentage"` $(y-\hat{y})/y$. Divides by the **observed target**. This is the per-observation contribution to the MAPE loss the epsilon-SVR models are fitted under, so it is the residual that corresponds to the estimated objective. `mean(abs(.)) * 100` is the training MAPE.

`"multiplicative"` $(y-\hat{y})/\hat{y}$. Divides by the **fitted value**. This is the $\hat{\eta}$ of the multiplicative-noise model $Y = f(x)(1+\eta)$, under which $\eta = (Y - f(x))/f(x)$. Use it to inspect the assumed noise structure, e.g. checking whether $\hat{\eta}$ is homoscedastic and centred at zero.

The denominators differ and the choice matters: `"percentage"` divides by the observed target because that is what the MAPE loss does, `"multiplicative"` divides by the fitted value because that is what the noise model does. They agree only when $y = \hat{y}$, and diverge as the fit degrades. Choose by the question being asked: the loss actually minimised (`"percentage"`), or the noise model assumed (`"multiplicative"`).

Value

Numeric vector of length `N` (the number of training observations), in training-row order.

Near-zero fitted values

`"multiplicative"` divides by $\hat{y}$, which an SVR does not constrain to be positive even though the targets are. Where $\hat{y}$ is at or below `sqrt(.Machine$double.eps) * mean(abs(y))` - including zero and negative fitted values - the ratio is inflated, infinite, or sign-flipped.

This function does **not** drop those observations: it always returns exactly `N` values in training-row order, so the result stays aligned with `y_train`, `fitted()`, and the training rows. The raw value (possibly `Inf` or `NaN`) is returned and a single warning reports how many observations are affected.

For *pooled* diagnostics (a mean multiplicative error, a variance, a histogram) the affected observations must be excluded, or one near-zero fitted value dominates the summary. This follows the standard treatment of percentage errors near zero (Makridakis): screen the fitted values against a threshold and drop the observations below it before pooling, rather than altering the per-observation values. Exclude at the point of aggregation, e.g.:

```
e <- residuals(fit, type = "multiplicative")
mean(e[is.finite(e)])
```

Not reachable through parsnip

parsnip registers neither `residuals.model_fit` nor `fitted.model_fit`, so calling either generic on a `model_fit` dispatches to the stats default and returns `NULL` **silently** - no error, no warning. Reach the psvr object first with `parsnip::extract_fit_engine()`, then call `residuals()` or `fitted()` on that. psvr deliberately does not register S3 methods on parsnip's class. Note also that `parsnip::augment()` recomputes predictions on whatever `new_data` it is given and reports response residuals only, so it is not a substitute for the `"percentage"` and `"multiplicative"` types.

See Also

`fitted.psvr_mape()` and the other fitted methods

Examples

```
set.seed(1)
X <- matrix(runif(40, 0.5, 3), 20, 2)
y <- 2 + X[, 1]^2
fit <- psvr_rmspe(X, y, kernel = make_kernel("rbf"), gamma = 100)
head(residuals(fit))
mean(abs(residuals(fit, type = "percentage"))) * 100   # training MAPE

# Through parsnip both generics return NULL on the model_fit wrapper;
# extract the engine object first.
df   <- data.frame(x1 = X[, 1], x2 = X[, 2], y = y)
spec <- psvr_rmspe_rbf(cost = 10, rbf_sigma = 0.8)
pfit <- parsnip::fit(spec, y ~ x1 + x2, data = df)
residuals(pfit)                                    # NULL
```

```
head(residuals(parsnip::extract_fit_engine(pfit)))  # the residuals
```

---

psvr_cv *Cross-validate psvr_mape() with automatic warm-start across folds*

Description

Fits a psvr_mape() model on each split in splits, carrying the converged dual variables (alpha, alpha_star) from one fold into the next as the SMO warm-start, so each solve starts from the previous fold's optimum instead of from zero. Because consecutive folds share most of their training rows, that starting point is already close to feasible; before each solve the carried vectors are projected back onto the constraint set (the equality $\sum_k \beta_k = 0$ and the per-sample box), with the residual violation absorbed by the rows that are new to this fold. The warm-start procedure is Algorithm 1 of arXiv:2605.01446 v3. Returns a tibble with one row per fold.

Usage

```
psvr_cv(
  splits,
  ...,
  X_var = NULL,
  y_var = NULL,
  warm_start = TRUE,
  verbose = FALSE
)
```

Arguments

splits	Either an rsample::rset object (e.g. from rsample::vfold_cv()), or a list of named lists each containing analysis (data frame), assessment (data frame), and optionally row_ids (integer vector of original training-row indices used for warm-start alignment across folds; defaults to positional).
...	Arguments forwarded to psvr_mape(). Must specify kernel and the MAPE hyperparameters (C, eps). alpha_init and alpha_star_init are managed internally; supplying them via ... is an error, and so is loss, which is not an argument of psvr_mape().
X_var	Character vector of predictor column names.
y_var	Single character giving the target column name.
warm_start	Logical; if FALSE, each fold fits cold-start (useful for benchmarking the T5 speedup).
verbose	Logical; if TRUE, report per-fold progress via message() (suppressible with suppressMessages()).

Details

This helper is **MAPE-only**, and there is no `loss` argument. That is a limitation of the implementation, not of the method: only `psvr_mape()` was ever wired to it. LS-SVR cross-validates perfectly well, it simply has no carryover state to exploit (each fold is a single linear-system solve), so for `psvr_rmspe()` use `tune::tune_grid()` with parallel cold-start.

Value

A `tibble` with one row per split and columns:

`split_id` 1-based fold index.

`fit` A list-column of `psvr_mape` objects, or `psvr_mape_sym` when `sym_type` is `"even"` or `"odd"`.

`predictions` A list-column of numeric vectors (predictions on the assessment set).

`metrics` A list-column of named numeric vectors (`mape`, `rmspe`, `mse`, `r2`).

`iter_count` Integer; SMO iterations from `fit$iterations`.

`elapsed_sec` Numeric; wall-clock seconds for the fit.

`warm_started` Logical; `TRUE` for fold > 1 when `warm_start = TRUE`.

Examples

```
if (requireNamespace("rsample", quietly = TRUE) &&
    requireNamespace("tibble",  quietly = TRUE)) {
  set.seed(2026)
  d <- data.frame(
    y  = stats::rlnorm(80, sdlog = 1.0),
    x1 = stats::rnorm(80),
    x2 = stats::rnorm(80)
  )
  folds <- rsample::vfold_cv(d, v = 5)
  res <- psvr_cv(folds, X_var = c("x1", "x2"), y_var = "y",
                 kernel = make_kernel("rbf", sigma = 1),
                 C = 10, eps = 5)
  median(vapply(res$metrics, function(m) m[["mape"]], numeric(1)))
}
```

`psvr_mape` *Fit an epsilon-SVR with MAPE loss*

Description

Fits the percentage-error epsilon-SVR of the paper: Model 1 when `sym_type = "none"`, and the symmetric-kernel Model 2 when `sym_type` is `"even"` or `"odd"`. The dual is a quadratic program with the sample-dependent box $|\beta_k| \le 100C/y_k$ and $\sum_k \beta_k = 0$, solved by the built-in SMO loop or by `osqp`.

Usage

```
psvr_mape(
  X,
  y,
  sym_type = c("none", "even", "odd"),
  kernel,
  C,
  eps,
  solver = c("smo", "osqp"),
  tol = 0.001,
  max_iter = 100000L,
  alpha_init = NULL,
  alpha_star_init = NULL,
  warm_start_check = TRUE,
  block_k4_enabled = TRUE,
  engine = c("rcpp", "r"),
  ...,
  alpha_couple = 0.5,
  precomputed_Omega = NULL,
  precomputed_Omega_s = NULL
)
```

Arguments

X	Numeric matrix of training inputs, one observation per row ($N \times p$).
y	Numeric vector of training targets, length N. Must satisfy $y_k > 0$ for every k; percentage-error loss is undefined otherwise, and this is checked rather than coerced.
sym_type	Symmetry type, one of "none" (default), "even" or "odd". Maps onto the symmetry parameter a of the paper: "none" fits Model 1 and imposes no symmetry constraint; "even" sets $a = +1$, enforcing $f(x) = f(-x)$; "odd" sets $a = -1$, enforcing $f(x) = -f(-x)$. This is the same vocabulary as the sym_type argument of the parsnip specifications, so the two public surfaces agree. The symmetric variants require a kernel satisfying Assumption 3 of the paper – see make_kernel().
kernel	A kernel function created by make_kernel().
C	Regularization parameter, $C > 0$. Required.
eps	Insensitivity tube half-width in percentage units, $\epsilon \geq 0$. Required.
solver	Backend for the dual quadratic program, "smo" (default) or "osqp". See the section above.
tol	Numerical tolerance, default 1e-3. **Its meaning depends on** solver**.** Under "smo" it is the convergence tolerance of the SMO loop, and tighter values produce more iterations. Under "osqp" the solver runs at its own fixed tolerances and tol is used only afterwards, as the threshold below which a dual variable counts as zero when identifying free, saturated and support-vector sets.

`max_iter` Maximum SMO iterations, default `100000L`. The solver emits a `warning()` and returns `converged = FALSE` if it does not converge within `max_iter`. Ignored for `solver = "osqp"`.

`alpha_init, alpha_star_init`
Optional warm-start vectors for the SMO solver, each a finite numeric vector of length N. Projected onto $\sum_k(\alpha_k - \alpha_k^*) = 0$ intersected with the per-sample box $[0, 100C/y_k]$ before the solve. `NULL` (default) cold-starts. `psvr_cv()` manages these automatically across folds.

`warm_start_check`
Logical; if `TRUE` (default), validate post-projection feasibility and `stop()` on violation. A surviving equality residual is fatal rather than cosmetic: SMO conserves $\sum_k(\alpha_k - \alpha_k^*)$, so an infeasible start is carried through to the returned solution.

`block_k4_enabled`
Logical; if `TRUE` (default), enable the block-k=4 SMO inner loop. Each outer iteration may select a second working pair and apply a two-dimensional joint update when the descent-guaranteed decoupling criterion holds. `FALSE` restores the k=2 behaviour bit-identically.

`engine` One of `"rcpp"` (default) or `"r"`. Selects the SMO backend: the C++ core, or the R reference implementation. Both produce bit-identical results on the development toolchain; `"r"` is retained as the reference and will be deprecated in 0.0.4.0 and removed in 0.1.0.

`...` Must be empty. Present only so that `alpha_couple`, `precomputed_Omega` and `precomputed_Omega_s` must be matched by their exact names rather than by position or partial matching – without it `precomputed_Omega` would be a partial-match prefix of `precomputed_Omega_s`. Passing anything here is an error, which is how a mistyped argument name is caught.

`alpha_couple` Numeric in $[0, 1]$, default `0.5`. Coupling penalty in the second-pair selection score $\text{gain} \times (1 - \alpha_{\text{couple}} \cdot \text{coupling})$. Exposed for empirical tuning; rarely needs adjustment. Ignored when `block_k4_enabled = FALSE`.

`precomputed_Omega, precomputed_Omega_s`
INTERNAL – used by `psvr_cv()` to share one full-dataset kernel matrix across folds. Users should not set these. `precomputed_Omega` applies when `sym_type = "none"`, `precomputed_Omega_s` otherwise.

Details

For the least-squares / RMSPE family (Models 3 and 4) see `psvr_rmspe()`. The two are deliberately separate functions: they share no solver, no dual structure and no hyperparameter search space, so a single signature would make most of its own arguments conditional. The name `psvr()` is reserved for a future automatic-selection front end and is **not** a synonym for either.

Value

For `sym_type = "none"`, an object of class `"psvr_mape"`: a list with components `beta` (the pruned dual differences $\beta = \alpha - \alpha^*$ over the support-vector indices, used by `predict()`), `alpha` and `alpha_star` (the length-N pre-pruning duals, retained for warm starts), `b`, `X_sv`, `y_sv`, `y_train`, `fitted_values`, `kernel`, `C`, `eps`, `n_train`, `p_train`, `iterations`, `converged` and `block_k4`.

For `sym_type = "even"` or `"odd"`, an object of class `"psvr_mape_sym"`: the same components plus `a` (the symmetry parameter) and `spectral` (adaptive spectral-shift diagnostics).

Methods are available for `predict()`, `print()`, `coef()`, `summary()`, `fitted()` and `residuals()`.

Choosing a solver

`solver = "smo"` (the default) uses the built-in sequential minimal optimisation loop. On some problems it does **not** converge within `max_iter`: it emits a `warning()`, `converged` is `FALSE` and `iterations` reaches the cap. This was first observed with linear and polynomial kernels, but it also occurs with the RBF kernel, so the kernel is **not** what determines it, and what does has not been established. Check `converged` on the returned fit, and refit with `solver = "osqp"` when it is `FALSE` and accuracy matters.

See Also

`psvr_rmspe()` for the LS-SVR / RMSPE family, `psvr_cv()` for cross-validation with warm-start carryover, `make_kernel()` for kernels.

Examples

```
set.seed(1)
X <- matrix(rnorm(40), 20, 2)
y <- rlnorm(20)
K <- make_kernel("rbf", sigma = 1)

fit <- psvr_mape(X, y, kernel = K, C = 10, eps = 5)
predict(fit, X[1:3, , drop = FALSE])

# Even-symmetric variant (Model 2): f(x) = f(-x).
fit_sym <- psvr_mape(X, y, sym_type = "even", kernel = K, C = 10, eps = 5)
predict(fit_sym, X[1:3, , drop = FALSE])
```

---

psvr_mape_specs *Parsnip model specs: epsilon-SVR with MAPE loss (Model 1)*

Description

Create parsnip model specifications for `psvr_mape()` with a fixed kernel type. Kernel parameters are tunable parsnip arguments; the symmetry parameter `a` and solver tolerance are engine arguments passed via `set_engine()`.

Usage

```
psvr_mape_rbf(
  mode = "regression",
  engine = "psvr",
  cost = NULL,
```

```
  margin = NULL,
  rbf_sigma = NULL,
  sym_type = NULL
)

psvr_mape_poly(
  mode = "regression",
  engine = "psvr",
  cost = NULL,
  margin = NULL,
  degree = NULL,
  scale_factor = NULL,
  sym_type = NULL
)

psvr_mape_linear(
  mode = "regression",
  engine = "psvr",
  cost = NULL,
  margin = NULL,
  sym_type = NULL
)
```

Arguments

mode	Only "regression" is supported.
engine	Only "psvr" is available.
cost	Regularization parameter C > 0. Use hardhat::tune() to optimize. Mapped to cost_psvr(), whose default range is [-2, 10] on the log2 scale (about 0.25 to 1024). That range is adequate here, where typical ϵ-SVR optima lie in [10, 100]. The LS-SVR specs (psvr_rmspe_specs) map cost to Γ and need a much wider range; see there.
margin	Epsilon tube half-width $\epsilon \geq 0$ expressed as a percentage of each target value. Use hardhat::tune() to optimize. Mapped to margin_percentage() with default range [1, 20] (percentage units). Named to match parsnip::svm_rbf(); note the units differ from dials::svm_margin(), which is absolute rather than percentage.
rbf_sigma	RBF bandwidth $\sigma > 0$. Use hardhat::tune() to optimize. Mapped to rbf_sigma_psvr(), whose default range [-3, 1] on the log10 scale is a fixed, conservative fallback. **The range does not finalize automatically from the training data.** rbf_sigma_psvr() sets finalize = NULL, so dials::finalize() leaves it untouched. To centre the range on the data, pass rbf_sigma_psvr_data() computed on the **preprocessed** predictors explicitly — via update() on the extracted parameter set, or via psvr_option_add() for a workflow set. (RBF specs only.)
sym_type	Symmetry type: "none" (default) fits the non-symmetric ϵ-SVR of Model 1; "even" (a = 1) and "odd" (a = -1) fit the symmetric ϵ-SVR of Model 2. Use

	hardhat::tune() to optimise over the levels during CV; see sym_type_param() to restrict which levels are searched.
degree	Polynomial degree ≥ 1. Use hardhat::tune() to optimize. (Polynomial specs only.)
scale_factor	Polynomial constant term (coef0). Use hardhat::tune() to optimize. (Polynomial specs only.)

Value

A parsnip model_spec object of the corresponding class.

Examples

```
library(parsnip)
spec <- psvr_mape_rbf(cost = 10, margin = 1, rbf_sigma = 1) |>
  set_engine("psvr")

spec_poly <- psvr_mape_poly(cost = 10, margin = 1, degree = 2,
                            scale_factor = 1) |>
  set_engine("psvr")

spec_lin <- psvr_mape_linear(cost = 10, margin = 1) |>
  set_engine("psvr")

# Symmetric epsilon-SVR (Model 2) via the sym_type argument:
spec_sym <- psvr_mape_rbf(cost = 10, margin = 1, rbf_sigma = 1,
                          sym_type = "even") |>
  set_engine("psvr")
```

---

psvr_option_add *Apply data-driven rbf_sigma to all psvr workflows in a workflow set*

Description

A convenience wrapper that calls workflowsets::option_add() for every psvr workflow in wf_set (those whose wflow_id contains "m1", "m2", "m3", or "m4"), replacing the rbf_sigma dials parameter with a data-driven one built from X via rbf_sigma_psvr_data().

Usage

```
psvr_option_add(wf_set, X, width = 10, sample_size = 500L, seed = NULL)
```

Arguments

wf_set	A workflow_set object.
X	A numeric matrix or data frame of preprocessed predictors.
width	Positive scalar. Passed to rbf_sigma_psvr_data(). Default 10.

sample_size	Integer. Passed to sigma_heuristic(). Default 500L.
seed	Integer seed for subsampling. Default NULL.

Value

The updated workflow_set (the same object with option_add() applied to each psvr workflow).

See Also

rbf_sigma_psvr_data(), sigma_heuristic()

Examples

```
## Not run:
# After building wf_set and preprocessing:
train_baked <- rec |> prep() |> bake(new_data = train)
wf_set <- psvr_option_add(wf_set, train_baked |> select(-outcome))

## End(Not run)
```

---

psvr_option_add_cost_ls

Apply data-driven LS-SVR cost range to all m3/m4 workflows in a workflow set

---

Description

A convenience wrapper that calls workflowsets::option_add() for every LS-SVR psvr workflow in wf_set (those whose wflow_id matches "m3" or "m4"), replacing the cost dials parameter with one built from y via cost_psvr_ls_data().

Usage

```
psvr_option_add_cost_ls(wf_set, y, width_log2 = 4)
```

Arguments

wf_set	A workflow_set object.
y	Numeric vector of strictly positive training targets.
width_log2	Passed to cost_psvr_ls_data(). Default 4.

Details

Workflows for `m1`/`m2` (ϵ-SVR) are intentionally skipped — for those, `cost` maps to `C` and the static `cost_psvr()` range is usually adequate.

Note: `workflowsets::option_add()` replaces the whole `param_info` option for each matched workflow. If you also need a data-driven `rbf_sigma` (via `psvr_option_add()`), build the full `param_info` manually with `tune::extract_parameter_set_dials()` and call `workflowsets::option_add()` in one shot, or call this helper first and then `psvr_option_add()` (which only touches `rbf_sigma`) — the latter currently overwrites the former, so prefer the manual one-shot approach when both are needed.

Value

The updated `workflow_set`.

See Also

`cost_psvr_ls_data()`, `psvr_option_add()`

Examples

```
## Not run:
# After building wf_set with at least one m3 or m4 workflow:
wf_set <- psvr_option_add_cost_ls(wf_set, y = train_df$y)

## End(Not run)
```

`psvr_rmspe` *Fit a least-squares SVR with RMSPE loss*

Description

Fits the percentage-error LS-SVR of the paper: Model 3 when `sym_type = "none"`, and the symmetric-kernel Model 4 when `sym_type` is `"even"` or `"odd"`. There is no quadratic program and no sparsity: the fit is a single solve of the $(N+1) \times (N+1)$ augmented linear system

$$\begin{pmatrix} 0 & 1^\top \\ 1 & \Omega + Y_\Gamma \end{pmatrix} \begin{pmatrix} b \\ \alpha \end{pmatrix} = \begin{pmatrix} 0 \\ y \end{pmatrix}$$

with $Y_\Gamma = \mathrm{diag}(y_1^2/\Gamma, \ldots, y_N^2/\Gamma)$, and Ω_s replacing Ω in the symmetric case. Every training point contributes to the prediction.

Usage

```
psvr_rmspe(
  X,
  y,
  sym_type = c("none", "even", "odd"),
  kernel,
  gamma,
  precondition = "auto",
  ...
)
```

Arguments

X	Numeric matrix of training inputs, one observation per row ($N \times p$).
y	Numeric vector of training targets, length N. Must satisfy $y_k > 0$ for every k; percentage-error loss is undefined otherwise, and this is checked rather than coerced.
sym_type	Symmetry type, one of "none" (default), "even" or "odd". Maps onto the symmetry parameter a of the paper: "none" fits Model 3 and imposes no symmetry constraint; "even" sets $a = +1$, enforcing $f(x) = f(-x)$; "odd" sets $a = -1$, enforcing $f(x) = -f(-x)$. This is the same vocabulary as the sym_type argument of the parsnip specifications, so the two public surfaces agree. The symmetric variants require a kernel satisfying Assumption 3 of the paper – see make_kernel().
kernel	A kernel function created by make_kernel().
gamma	Regularization parameter $\Gamma > 0$. Required. Larger values weight the squared percentage residuals more heavily against the norm penalty.
precondition	One of "auto" (default), "always", "never", or a positive numeric threshold. Controls a symmetric rescaling of the linear system by $P = \mathrm{diag}(1/y)$. Unpreconditioned, the system carries the target-weighted diagonal y_k^2/Γ, whose entries span the square of the target range; solving $P\Omega P$ instead replaces it with the constant $1/\Gamma$, so the diagonal no longer inflates the condition number when the targets differ by orders of magnitude. The multipliers are rescaled back on the way out, leaving the solution unchanged in exact arithmetic. "auto" applies it when the target ratio $\max(y)/\min(y)$ exceeds 10; a numeric value sets that threshold explicitly. Whether it fired is reported in fit$precondition_applied.
...	Must be empty. Passing anything here is an error, which is how a mistyped argument name is caught.

Details

For the epsilon-SVR / MAPE family (Models 1 and 2) see psvr_mape(). The two are deliberately separate functions: they share no solver, no dual structure and no hyperparameter search space. The name psvr() is reserved for a future automatic-selection front end and is **not** a synonym for either.

Value

For `sym_type = "none"`, an object of class `"psvr_rmspe"`: a list with components `alpha` (the length-N multipliers), `b`, `X_train`, `y_train`, `fitted_values`, `kernel`, `gamma`, `n_train`, `p_train` and `precondition_applied`.

For `sym_type = "even"` or `"odd"`, an object of class `"psvr_rmspe_sym"`: the same components plus `a` (the symmetry parameter).

Methods are available for `predict()`, `print()`, `coef()`, `summary()`, `fitted()` and `residuals()`. Note that `coef()` returns three components here (`alpha`, `b`, `support_data`) against five for the MAPE classes: LS-SVR has no `alpha_star` and no pruned `beta`, and the absent components are not materialised as `NULL`.

See Also

`psvr_mape()` for the epsilon-SVR / MAPE family, `make_kernel()` for kernels.

Examples

```
set.seed(1)
X <- matrix(rnorm(40), 20, 2)
y <- rlnorm(20)
K <- make_kernel("rbf", sigma = 1)

fit <- psvr_rmspe(X, y, kernel = K, gamma = 100)
predict(fit, X[1:3, , drop = FALSE])

# Even-symmetric variant (Model 4): f(x) = f(-x).
fit_sym <- psvr_rmspe(X, y, sym_type = "even", kernel = K, gamma = 100)
predict(fit_sym, X[1:3, , drop = FALSE])
```

---

psvr_rmspe_specs *Parsnip model specs: LS-SVR with RMSPE loss (Model 3)*

Description

Create parsnip model specifications for `psvr_rmspe()` with a fixed kernel type. `cost` maps to the regularization parameter Γ.

Usage

```
psvr_rmspe_rbf(
  mode = "regression",
  engine = "psvr",
  cost = NULL,
  rbf_sigma = NULL,
  sym_type = NULL
)
```

```
psvr_rmspe_poly(
  mode = "regression",
  engine = "psvr",
  cost = NULL,
  degree = NULL,
  scale_factor = NULL,
  sym_type = NULL
)

psvr_rmspe_linear(
  mode = "regression",
  engine = "psvr",
  cost = NULL,
  sym_type = NULL
)
```

Arguments

mode	Only "regression" is supported.
engine	Only "psvr" is available.
cost	Regularization parameter $\Gamma > 0$. Use hardhat::tune() to optimize. Mapped to cost_psvr(), whose default range [-2, 10] on the log2 scale ($\Gamma \leq 1024$) is the ϵ-SVR range and is **too narrow for LS-SVR**. Γ enters the LS-SVR system only through the y_k^2/Γ diagonal, so the value that balances that term against the kernel scales with the outcome's variance and with the sample size — the quantity cost_psvr_ls_data() computes. The LS-SVR optimum is therefore routinely orders of magnitude above the static ceiling, and a grid over the default is boundary-trapped whenever it is. Pass cost_psvr_ls_data() built from the training outcome explicitly, via update() on the extracted parameter set or via psvr_option_add_cost_ls() for a workflow set. This **cannot** be automated: tune finalizes parameters from the molded **predictors** only and never passes the outcome to dials::finalize(), so no finalize function on cost could compute it.
rbf_sigma	RBF bandwidth $\sigma > 0$. Use hardhat::tune() to optimize. Mapped to rbf_sigma_psvr(), whose default range [-3, 1] on the log10 scale is a fixed, conservative fallback. **The range does not finalize automatically from the training data.** rbf_sigma_psvr() sets finalize = NULL, so dials::finalize() leaves it untouched. To centre the range on the data, pass rbf_sigma_psvr_data() computed on the **preprocessed** predictors explicitly — via update() on the extracted parameter set, or via psvr_option_add() for a workflow set. (RBF specs only.)
sym_type	Symmetry type: "none" (default) fits the non-symmetric LS-SVR of Model 3; "even" (a = 1) and "odd" (a = -1) fit the symmetric LS-SVR of Model 4. Use hardhat::tune() to optimise over the levels during CV; see sym_type_param() to restrict which levels are searched.
degree	Polynomial degree ≥ 1. Use hardhat::tune() to optimize. (Polynomial specs only.)

scale_factor	Polynomial constant term (coef0). Use hardhat::tune() to optimize. (Polynomial specs only.)

Value

A parsnip model_spec object of the corresponding class.

Engine arguments

The precondition argument of psvr_rmspe() is exposed as a non-tunable engine argument. Pass it via parsnip::set_engine(), e.g. set_engine("psvr", precondition = "always"). Default is "auto". See psvr_rmspe() for accepted values and semantics.

Examples

```
library(parsnip)
spec <- psvr_rmspe_rbf(cost = 1000, rbf_sigma = 1) |>
  set_engine("psvr")

spec_poly <- psvr_rmspe_poly(cost = 1000, degree = 2, scale_factor = 1) |>
  set_engine("psvr")

spec_lin <- psvr_rmspe_linear(cost = 1000) |>
  set_engine("psvr")

# Symmetric LS-SVR (Model 4) via the sym_type argument:
spec_sym <- psvr_rmspe_rbf(cost = 1000, rbf_sigma = 1,
                           sym_type = "even") |>
  set_engine("psvr")
```

---

rbf_sigma_psvr *RBF sigma parameter for psvr models*

Description

A dials parameter for the RBF kernel bandwidth in psvr models. The default range [-3, 1] on the log10 scale is a conservative fallback. For best results, override the range using sigma_heuristic() computed on the preprocessed training data:

Usage

```
rbf_sigma_psvr(range = c(-3, 1), trans = scales::log10_trans())
```

Arguments

range	Numeric vector of length 2 on the log10 scale. Default c(-3, 1).
trans	A scales transformation object. Default scales::log10_trans().

Details

```
train_baked <- rec |> prep() |> bake(new_data = train)
sigma_med   <- sigma_heuristic(train_baked |> select(-outcome))
rbf_sigma_custom <- rbf_sigma_psvr(
  range = c(log10(sigma_med / 10), log10(sigma_med * 10))
)
# Then inject via option_add():
wf_set |> option_add(
  param_info = extract_parameter_set_dials(wf) |>
    update(rbf_sigma = rbf_sigma_custom),
  id = "your_workflow_id"
)
```

Value

A quant_param dials object.

See Also

sigma_heuristic()

Examples

```
rbf_sigma_psvr()

# Override with data-driven range:
X <- matrix(rnorm(200), ncol = 4)
sigma_med <- sigma_heuristic(X)
rbf_sigma_psvr(range = c(log10(sigma_med / 10), log10(sigma_med * 10)))
```

---

rbf_sigma_psvr_data *RBF sigma parameter with data-driven range for psvr models*

Description

A convenience wrapper that combines sigma_heuristic() and rbf_sigma_psvr() in a single call. It computes the median pairwise distance from X and returns a quant_param whose search range spans one order of magnitude either side of the heuristic value on the log10 scale (i.e., [log10(sigma_med / width), log10(sigma_med * width)]).

Usage

```
rbf_sigma_psvr_data(
  X,
  width = 10,
  sample_size = 500L,
  seed = NULL,
```

```
  trans = scales::log10_trans()
)
```

Arguments

X	A numeric matrix or data frame of predictors (already preprocessed — centred, scaled, etc.).
width	Positive scalar. Multiplier that sets the half-width of the search range around the heuristic sigma. Default 10 (one decade).
sample_size	Integer. Passed to sigma_heuristic(). Default 500L.
seed	Integer seed for subsampling. Passed to sigma_heuristic(). Default NULL.
trans	A scales transformation object. Default scales::log10_trans().

Value

A quant_param dials object with a data-driven search range.

See Also

sigma_heuristic(), rbf_sigma_psvr()

Examples

```
X <- matrix(rnorm(200), ncol = 4)
rbf_sigma_psvr_data(X)
rbf_sigma_psvr_data(X, width = 5)
```

---

sigma_heuristic *Median-distance heuristic for RBF kernel bandwidth*

Description

Returns the median pairwise Euclidean distance between rows of X, which is a standard data-driven starting point for the RBF kernel bandwidth (Schölkopf & Smola, 2002). Use the result to define a sensible rbf_sigma search range centred on this value via dials::rbf_sigma(range = c(log10(sigma / 10), log10(sigma * 10))).

Usage

```
sigma_heuristic(X, sample_size = 500L, seed = NULL)
```

Arguments

X	A numeric matrix or data frame of predictors (already preprocessed — centred, scaled, etc.).
sample_size	Integer. If nrow(X) > sample_size, a random subsample is used to avoid O(n²) memory cost on large datasets. Default 500L.
seed	Integer seed for the subsample. Default NULL. The caller's RNG stream is restored on exit, so passing seed makes the subsample reproducible without affecting the caller's subsequent random draws.

Value

A scalar numeric: the median pairwise Euclidean distance.

Examples

```
X <- matrix(rnorm(200), ncol = 4)
sigma_heuristic(X)
```

---

summary.psvr_mape *Summarize a fitted epsilon-SVR with MAPE loss*

Description

Prints the kernel, the training and support-vector counts, the hyperparameters, and the SMO iteration count with its convergence status. Every training point contributes for LS-SVR but not here: the support-vector percentage is the sparsity of the fit.

Usage

```
## S3 method for class 'psvr_mape'
summary(object, ...)
```

Arguments

object	An object of class "psvr_mape", from psvr_mape() with sym_type = "none", or from a parsnip fit unwrapped with parsnip::extract_fit_engine().
...	Ignored.

Value

object, invisibly. Called for the printed summary.

See Also

print.psvr_mape(), coef.psvr_mape()

Examples

```
set.seed(1)
X <- matrix(rnorm(40), 20, 2)
y <- rlnorm(20)
fit <- psvr_mape(X, y, kernel = make_kernel("rbf", sigma = 1),
                 C = 10, eps = 5)
summary(fit)
```

`summary.psvr_mape_sym` *Summarize a fitted symmetric epsilon-SVR with MAPE loss*

Description

As `summary.psvr_mape()`, with the symmetry parameter a reported.

Usage

```
## S3 method for class 'psvr_mape_sym'
summary(object, ...)
```

Arguments

`object`	An object of class `"psvr_mape_sym"`, from `psvr_mape()` with `sym_type = "even"` or `"odd"`.
`...`	Ignored.

Value

`object`, invisibly. Called for the printed summary.

See Also

`print.psvr_mape_sym()`, `coef.psvr_mape_sym()`

Examples

```
set.seed(1)
X <- matrix(rnorm(40), 20, 2)
y <- rlnorm(20)
fit <- psvr_mape(X, y, sym_type = "even",
                 kernel = make_kernel("rbf", sigma = 1), C = 10, eps = 5)
summary(fit)
```

summary.psvr_rmspe *Summarize a fitted LS-SVR with RMSPE loss*

Description

Prints the kernel, the training count, the hyperparameter, and whether the `diag(1/y)` preconditioner fired. No support-vector count is reported: LS-SVR performs no pruning, so every training point contributes to the prediction.

Usage

```
## S3 method for class 'psvr_rmspe'
summary(object, ...)
```

Arguments

`object`	An object of class `"psvr_rmspe"`, from `psvr_rmspe()` with `sym_type = "none"`, or from a parsnip fit unwrapped with `parsnip::extract_fit_engine()`.
`...`	Ignored.

Value

`object`, invisibly. Called for the printed summary.

See Also

`print.psvr_rmspe()`, `coef.psvr_rmspe()`

Examples

```
set.seed(1)
X <- matrix(rnorm(40), 20, 2)
y <- rlnorm(20)
fit <- psvr_rmspe(X, y, kernel = make_kernel("rbf", sigma = 1), gamma = 100)
summary(fit)
```

summary.psvr_rmspe_sym *Summarize a fitted symmetric LS-SVR with RMSPE loss*

Description

As `summary.psvr_rmspe()`, with the symmetry parameter a reported.

Usage

```
## S3 method for class 'psvr_rmspe_sym'
summary(object, ...)
```

Arguments

object	An object of class "psvr_rmspe_sym", from psvr_rmspe() with sym_type = "even" or "odd".
...	Ignored.

Value

object, invisibly. Called for the printed summary.

See Also

print.psvr_rmspe_sym(), coef.psvr_rmspe_sym()

Examples

```
set.seed(1)
X <- matrix(rnorm(40), 20, 2)
y <- rlnorm(20)
fit <- psvr_rmspe(X, y, sym_type = "even",
                  kernel = make_kernel("rbf", sigma = 1), gamma = 100)
summary(fit)
```

---

sym_type_param *Dials parameter for symmetry type*

Description

Returns a qualitative dials::new_qual_param() describing the sym_type argument of the psvr model specs. "none" fits the non-symmetric model; "even" maps to a = 1L (standard symmetric kernel); "odd" maps to a = -1L (anti-symmetric kernel).

Usage

```
sym_type_param(values = c("none", "even", "odd"))
```

Arguments

values	Character vector of levels to search over. Any subset of c("none", "even", "odd"); defaults to all three. Pass values = c("even", "odd") to tune over the symmetric models only, which reproduces the two-level grid offered before psvr 0.0.2.9011.

Value

A qual_param object.

Examples

```
sym_type_param()
sym_type_param(values = c("even", "odd"))
```

Index