

Package ‘steinsampling’

October 10, 2026

Title Kernelized Stein Discrepancy for Goodness-of-Fit Tests and Stein Sampling

Version 0.1.3

Date 2026-10-06

Description Provides Stein-discrepancy goodness-of-fit tests and Stein-method-based sampling tools. The tests include kernel Stein discrepancy U- and V-statistics following Liu et al. (2016) [doi:10.48550/arXiv.1602.03253](https://doi.org/10.48550/arXiv.1602.03253) and Chwialkowski et al. (2016) [doi:10.48550/arXiv.1602.02964](https://doi.org/10.48550/arXiv.1602.02964), plus the finite set Stein discrepancy test of Jitkrittum et al. (2017) [doi:10.48550/arXiv.1705.07673](https://doi.org/10.48550/arXiv.1705.07673). The sampling tools include Stein thinning, Stein Points, Stein Point Markov chain Monte Carlo, and Stein variational gradient descent following Riabiz et al. (2022) [doi:10.48550/arXiv.2005.03952](https://doi.org/10.48550/arXiv.2005.03952), Chen et al. (2018) [doi:10.48550/arXiv.1803.10161](https://doi.org/10.48550/arXiv.1803.10161), Chen et al. (2019) [doi:10.48550/arXiv.1905.03673](https://doi.org/10.48550/arXiv.1905.03673), and Liu and Wang (2016) [doi:10.48550/arXiv.1608.04471](https://doi.org/10.48550/arXiv.1608.04471). Gaussian mixture utilities are included for constructing example targets, simulation, density evaluation, and score callbacks.

URL <https://github.com/junhao7622/steinsampling>,
<https://arxiv.org/abs/2608.26450>

BugReports <https://github.com/junhao7622/steinsampling/issues>

License GPL (>= 2)

Encoding UTF-8

Depends R (>= 4.0)

Imports stats, mvtnorm (>= 0.9-9994)

Suggests testthat

Collate 'steinsampling-package.R' 'stein_helpers.R' 'kernel_classes.R'
'stein_points_alternative_kernel.R' 'bootstrap.R' 'gmm_model.R'
'ksd_u_test.R' 'ksd_v_test.R' 'fssd_test.R' 'svgd.R'
'stein_thinning.R' 'stein_points.R' 'stein_point_mcmc.R'

Config/testthat/edition 3

NeedsCompilation no

Config/roxygen2/version 8.0.0

Author Junhao Gao [aut, cre],
Ery Arias-Castro [aut]

Maintainer Junhao Gao <jug049@ucsd.edu>

Repository CRAN

Date/Publication 2026-10-10 06:50:02 UTC

Contents

steinsampling-package	3
compute_tau	4
custom_stein_kernel	5
eval_kernel	6
find_median_distance	8
fmin_grid	9
fssd_grad_kernel	11
fssd_opt_test	12
fssd_rand_test	14
fssd_statistic	16
fssd_test	18
get_score_evaluator	20
gmm	21
ksd_uq_matrix	22
ksd_u_statistic	24
ksd_u_test	25
ksd_v_statistic	27
ksd_v_test	29
mala	31
sp_mcmc	33
stein_codescent	36
stein_kernel	38
stein_kernel_imq_score	39
stein_kernel_inverse_log	40
stein_kernel_matrix	41
stein_points	42
stein_thinning	44
svgd	46

Index	49
--------------	-----------

steinsampling-package *steinsampling: Kernelized Stein Discrepancy for Goodness-of-Fit Tests and Stein Sampling*

Description

Score-based goodness-of-fit tests and Stein sampling tools: kernel Stein discrepancy (KSD) and finite set Stein discrepancy (FSSD) tests, Stein thinning, Stein Points, Stein Point Markov chain Monte Carlo (SP-MCMC), Stein variational gradient descent (SVGD), and reusable Stein kernels.

Details

Many target distributions are specified by a density known up to a normalizing constant, making direct probability calculations and exact simulation difficult. Two related tasks arise: assessing whether observed data agree with the target distribution, and constructing a point set whose empirical distribution approximates it. `steinsampling` addresses both tasks using

$$s_p(x) = \nabla \log p(x),$$

where s_p is the target score.

For goodness-of-fit testing, the observations are compared with a specified target distribution through its score and a kernel. `ksd_u_test()` provides a KSD test for independent observations. `ksd_v_test()` supports independent observations and serially dependent observations under the corresponding assumptions. `fssd_test()` instead evaluates Stein features at a finite set of test locations, with random or optimized locations.

For sampling and compression, the choice depends on how points are to be obtained. `svgd()` moves an initial set of particles toward the target distribution. `stein_points()` constructs points by numerical optimization, whereas `sp_mcmc()` constructs points from short Markov chains. When samples are already available, `stein_thinning()` selects a representative sequence, allowing repeated selections.

These methods share a score-and-kernel setup. `stein_kernel()` provides the built-in kernels and links to custom kernel construction. Gaussian mixture targets can be constructed with `gmm()`, whose help page describes density evaluation, simulation, and score calculation.

The methods can be used as complete algorithms or assembled from individual components.

Author(s)

Maintainer: Junhao Gao <jug049@ucsd.edu>

Authors:

- Junhao Gao <jug049@ucsd.edu>
- Ery Arias-Castro <eariascastro@ucsd.edu>

See Also

Useful links:

- <https://github.com/junhao7622/steinsampling>
- <https://arxiv.org/abs/2608.26450>
- Report bugs at <https://github.com/junhao7622/steinsampling/issues>

compute_tau

Compute the FSSD feature matrix

Description

Builds the row-feature matrix used by `fssd_statistic()` and `fssd_null_pvalue()`. The rows of `V` represent the test locations $L = \{v_1, \dots, v_J\}$. For the complete goodness-of-fit test, see `fssd_test()`.

Usage

```
compute_tau(X, scores, V, kernel_obj)
```

Arguments

<code>X</code>	Numeric $\tilde{n} \times d$ sample matrix.
<code>scores</code>	Numeric $\tilde{n} \times d$ score matrix for <code>X</code> .
<code>V</code>	Numeric $J \times d$ matrix representing <code>L</code> , one test location per row.
<code>kernel_obj</code>	Stein kernel object. A Gaussian RBF kernel must have a fixed positive bandwidth.

Details

For observation x_i , target score $s_p(x_i)$, and test location v_j , define

$$\xi_p(x_i, v_j) = s_p(x_i)k(x_i, v_j) + \nabla_x k(x_i, v_j).$$

With $d = \text{ncol}(X)$ and $J = \text{nrow}(V)$, the returned row is

$$\tau_i = \tau(x_i) = \frac{1}{\sqrt{dJ}} \text{vec}\{\xi_p(x_i, v_1), \dots, \xi_p(x_i, v_J)\} \in \mathbb{R}^{dJ}.$$

Value

Numeric $\tilde{n} \times dJ$ matrix. Row `i` is `tau(x_i)`; columns are ordered by test location and then coordinate.

References

Jitkrittum, W., Xu, W., Szabó, Z., Fukumizu, K., and Gretton, A. (2017). A Linear-Time Kernel Goodness-of-Fit Test. *Advances in Neural Information Processing Systems*, 30, 262–271. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2017/hash/979d472a84804b9f647bc185a877a8b5-Ab.html.

See Also

`fssd_statistic()`, `fssd_null_pvalue()`

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
scores <- -X
V <- matrix(0, ncol = 1)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
compute_tau(X, scores, V, kernel)
```

custom_stein_kernel	Create a Stein kernel from callbacks
---------------------	--------------------------------------

Description

Creates a kernel object from user-supplied functions that compute the kernel value and its derivatives.

Usage

```
custom_stein_kernel(
  eval,
  grad_x,
  trace_mixed,
  fssd_grad = NULL,
  scale2 = NULL,
  precon = NULL,
  custom_grad_mode = c("analytic", "numeric")
)
```

Arguments

eval, grad_x, trace_mixed	Callbacks (k , X , Y , M), where k is the current kernel object and M is the preconditioning matrix.
fssd_grad	Optional analytic FSSD-opt callback (k , X , v_j , grads_X , g_block , M) returning grad_{v_j} and, when the kernel exposes a scale, grad_param .
scale2	Optional positive squared scale, read inside callbacks with <code>scale2_kernel(k)</code> .
precon	Optional symmetric positive-definite M .

custom_grad_mode
 "analytic" or location-only "numeric".

Details

eval, grad_x, and trace_mixed supply the kernel value, its first derivative, and its mixed-derivative trace. M may be NULL; callbacks implement any preconditioning they support. For $X \in \mathbb{R}^{n_X \times d}$ and $Y \in \mathbb{R}^{n_Y \times d}$, eval returns the $n_X \times n_Y$ matrix $k(X, Y)$, grad_x the $n_X \times n_Y \times d$ array $\nabla_x k(X, Y)$, and trace_mixed the $n_X \times n_Y$ matrix $\text{tr}\{\nabla_x \nabla_y^\top k(X, Y)\}$. Kernel symmetry supplies $\nabla_y k$ by reversing the arguments of grad_x. [eval_kernel\(\)](#), [grad_x_kernel\(\)](#), and [trace_mixed_kernel\(\)](#) evaluate these callbacks on supplied points. [stein_kernel_matrix\(\)](#) combines the kernel calculations with target scores to form the Stein-kernel matrix.

KSD uses all three callbacks; FSSD uses only eval and grad_x. For FSSD-opt a custom kernel must provide fssd_grad or use custom_grad_mode = "numeric". A custom kernel can update its scale when scale2 is supplied; that path requires fssd_grad, because numeric mode differentiates the test locations only. [fssd_grad_kernel\(\)](#) evaluates fssd_grad on supplied points, and [fssd_opt_test\(\)](#) uses the result to optimize the test locations.

Value

A SteinKernel object containing the supplied callbacks. Use `print(kernel)` to display the kernel type, parameters, and the dimensions of the preconditioning matrix, when present.

See Also

[stein_kernel\(\)](#) for built-in kernels.

Examples

```
rbf <- stein_kernel("gaussian_rbf", h = 1)
kernel <- custom_stein_kernel(
  eval = function(k, X, Y, M) eval_kernel(rbf, X, Y, precon = M),
  grad_x = function(k, X, Y, M) grad_x_kernel(rbf, X, Y, precon = M),
  trace_mixed = function(k, X, Y, M) trace_mixed_kernel(rbf, X, Y, precon = M)
)
X <- matrix(c(-1, 0, 1), ncol = 1)
all.equal(stein_kernel_matrix(kernel, X, -X),
  stein_kernel_matrix(rbf, X, -X))
```

eval_kernel

Evaluate a base kernel and its derivatives

Description

Returns the kernel value for each pair of points in X and Y, the kernel's derivatives with respect to the coordinates of each point in X, the mixed-derivative trace H , and the score-derivative coupling C .

Usage

```
eval_kernel(obj, X, Y = NULL, precon = NULL)

grad_x_kernel(obj, X, Y = NULL, precon = NULL)

trace_mixed_kernel(obj, X, Y = NULL, precon = NULL)

cross_kernel(obj, X, grads, Y = NULL, grads_Y = NULL, precon = NULL)
```

Arguments

`obj` A SteinKernel object providing the requested operation.

`X` Numeric $n_X \times d$ matrix.

`Y` Optional numeric $n_Y \times d$ matrix; NULL uses X.

`precon` Optional preconditioner overriding `obj$precon`.

`grads, grads_Y` Score matrices matching X and Y.

Details

`trace_mixed_kernel()` returns the matrix H with entries

$$H_{ij} = \sum_{r=1}^d \partial_{x_r} \partial_{y_r} k(x_i, y_j) = \text{tr}\{\nabla_x \nabla_y^\top k(x_i, y_j)\}.$$

`cross_kernel()` returns the matrix C with entries

$$C_{ij} = s_p(x_i)^\top \nabla_y k(x_i, y_j) + s_p(y_j)^\top \nabla_x k(x_i, y_j),$$

where $s_p(x) = \nabla \log p(x)$. It is derived from `grad_x` using $\nabla_y k(x, y) = \nabla_x k(y, x)$, which holds for every symmetric base kernel, so no separate callback is needed. For its role in the complete kernel calculation, see `stein_kernel_matrix()`.

Value

`eval_kernel()` returns the numeric $n_X \times n_Y$ matrix $B_{ij} = k(x_i, y_j)$. `grad_x_kernel()` returns the numeric $n_X \times n_Y \times d$ array $G_{ijr} = \partial k(x_i, y_j) / \partial x_r$. `trace_mixed_kernel()` returns the numeric $n_X \times n_Y$ matrix H . `cross_kernel()` returns the numeric $n_X \times n_Y$ matrix C .

See Also

`stein_kernel()` for kernel construction.

Examples

```
k <- stein_kernel("gaussian_rbf", h = 1)
X <- matrix(c(-1, 0, 1), ncol = 1)
eval_kernel(k, X)
G <- grad_x_kernel(k, X)
```

```

dim(G)      # n_X x n_Y x d
G[, , 1]
trace_mixed_kernel(k, X)
S <- -X
all.equal(stein_kernel_matrix(k, X, S),
          tcrossprod(S) * eval_kernel(k, X) + cross_kernel(k, X, S) +
          trace_mixed_kernel(k, X))

```

find_median_distance *Compute and set the squared kernel scale*

Description

[find_median_distance\(\)](#) computes the squared median Euclidean distance between sample rows. [scale2_kernel\(\)](#) returns a kernel's squared scale, or an updated kernel when a new value is supplied.

Usage

```

find_median_distance(Z)

scale2_kernel(obj, value = NULL)

```

Arguments

Z	Finite numeric vector, matrix, or data frame with at least two observations. Matrix rows are observations; a vector is treated as an $n \times 1$ sample.
obj	A SteinKernel object.
value	Optional new squared scale.

Details

For rows $z_1, \dots, z_n$, the scale returned by [find_median_distance\(\)](#) is

$$\rho_{\text{med}} = \left\{ \text{median}_{i < j} \|z_i - z_j\|_2 \right\}^2.$$

The median is taken before squaring, so ρ_{med} is a squared scale and enters as `scale2_kernel(obj, rho_med)`, not as an unsquared bandwidth. All $n(n-1)/2$ pairs are used, so the calculation requires quadratic time and memory. Coordinates are not standardized. If the median distance is zero, the function warns and returns the floor value 10^{-5} . For its use in test setup, see [ksd_u_test\(\)](#) and [ksd_v_test\(\)](#).

The squared scale read and set by [scale2_kernel\(\)](#) is h^2 for Gaussian RBF and c^2 for IMQ. It is NULL for a kernel that exposes no scale, and NA_real_ for one whose scale exists but is not set. For kernel construction and parameter choices, see [stein_kernel\(\)](#).

Value

`find_median_distance()` returns the positive scalar ρ_{med} . `scale2_kernel()` returns the squared scale, or the updated kernel when value is supplied.

Examples

```
k <- stein_kernel("gaussian_rbf", h = 2)
scale2_kernel(k) # 4 = h^2

k <- scale2_kernel(k, 9)
scale2_kernel(k) # 9

rho <- find_median_distance(c(0, 1, 3)) # 4 = median(c(1, 2, 3))^2
k <- scale2_kernel(k, rho)
scale2_kernel(k) # 4
```

fmin_grid

*Optimizers for Stein Points***Description**

Creates a candidate-search function for `stein_points()` and `stein_codescent()`. The returned optimizer selects the point with the smallest objective among grid points (`fmin_grid()`), random candidates in the box (`fmin_mc()`), or local solutions from several Nelder-Mead runs started in the box (`fmin_nm()`).

Usage

```
fmin_grid(lb, ub, n0 = 100, grow = TRUE)

fmin_mc(lb, ub, n_mc = 20, mu0 = NULL, Sigma0 = NULL, sigsq = 1, delay = 20)

fmin_nm(
  lb,
  ub,
  n_res = 3,
  mu0 = NULL,
  Sigma0 = NULL,
  sigsq = 1,
  delay = 20,
  control = list(reltol = 0.001)
)
```

Arguments

`lb, ub` Finite lower and upper bound vectors of equal length, with `ub > lb` component-wise.

n0	Positive integer grid size, recycled across dimensions, or a length-d vector of positive integer sizes.
grow	A single logical value; whether to increase grid size as points are selected.
n_mc	Positive integer number of Monte Carlo candidates.
mu0, Sigma0	Initial Gaussian proposal mean and covariance. NULL uses the box midpoint and diagonal variances $((ub - lb) / 4)^2$, respectively. Supplied values must be a finite length-d vector and a finite symmetric positive-definite $d \times d$ matrix.
sigsq	Finite positive local proposal variance after the delay period.
delay	Nonnegative integer number of optimization iterations before using local proposals.
n_res	Positive integer number of random restarts.
control	List passed to <code>stats::optim()</code> .

Details

With `grow = TRUE`, the `fmin_grid()` resolution increases as the point set grows: the per-dimension size is $n0 + \text{round}(\sqrt{t})$, where t is the optimizer iteration supplied by `stein_points()` or `stein_codescent()`. It is practical mainly in low dimension because the total number of candidates is the product of the grid sizes across dimensions.

With `fmin_mc()`, early iterations draw candidates from a broad Gaussian distribution truncated to $[lb, ub]$. Once t exceeds `delay`, the proposal becomes local: it chooses one of the current points and draws a Gaussian perturbation with variance `sigsq`, again keeping only candidates in the box.

The optimizer returned by `fmin_nm()` performs a multi-start local search. A sine-squared transformation maps unconstrained Nelder-Mead parameters back into $[lb, ub]$, so every objective evaluation stays inside the requested search box. Each restart begins from the same boxed proposal rule used by `fmin_mc()`.

Value

An optimizer function `function(f, X_curr, t = nrow(X_curr) + 1L)`, where f is the objective described in `stein_points()`. It returns a list with `x_min`, `d_min`, `f_min`, and `n_eval`. `d_min` is the target score at the selected point and `f_min` is its objective value. The searches differ in what they select and what `n_eval` counts:

- `fmin_grid()`: `x_min` is the grid row with the smallest objective, and `n_eval` is the number of grid rows scored.
- `fmin_mc()`: `x_min` is the best candidate row, and `n_eval` is the number of candidates scored.
- `fmin_nm()`: `x_min` is the selected point, and `n_eval` is the number of objective evaluations charged to the search.

References

Chen, W. Y., Mackey, L., Gorham, J., Briol, F.-X., and Oates, C. J. (2018). Stein Points. Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 80, 844–853. <https://proceedings.mlr.press/v80/chen18f.html>.

Examples

```
# Target: N(0, 1) in each coordinate.
set.seed(1)
lb <- c(-4, -4)
ub <- c(4, 4)
opts <- list(grid = fmin_grid(lb, ub, n0 = 21, grow = FALSE),
             mc = fmin_mc(lb, ub, n_mc = 200),
             nm = fmin_nm(lb, ub, n_res = 3))
vapply(opts, function(opt) {
  fit <- stein_points(function(X) -X, stein_kernel("img"),
                     n_points = 20, d = 2, optimizer = opt, x_init = c(0, 0))
  tail(fit$ksd, 1)
}, numeric(1))
```

fssd_grad_kernel

Compute objective gradients for one FSSD test location

Description

Tells an optimizer how moving one test location or changing the kernel scale changes its objective. The input `g_block` specifies how that objective changes with each Stein feature; `fssd_opt_test()` computes this input internally.

Usage

```
fssd_grad_kernel(obj, X, vj, grads_X, g_block, precon = NULL)
```

Arguments

<code>obj</code>	A <code>SteinKernel</code> object supporting FSSD gradients. A Gaussian RBF kernel must have a fixed positive bandwidth.
<code>X</code>	Numeric $n \times d$ matrix.
<code>vj</code>	Test location, a numeric vector of length d .
<code>grads_X</code>	Target-score matrix matching <code>X</code> .
<code>g_block</code>	Numeric $n \times d$ matrix of the derivatives defined below, one row per observation and one column per coordinate.
<code>precon</code>	Optional preconditioner overriding <code>obj\$precon</code> .

Details

Let C be the objective computed from the Stein features

$$\xi_p(x_i, v_j) = s_p(x_i)k(x_i, v_j) + \nabla_x k(x_i, v_j),$$

where s_p is the target score. Supply $\partial C / \partial \xi_{p,r}(x_i, v_j)$ in `g_block[i, r]`. The function applies the chain rule to obtain the derivatives of C with respect to v_j and the squared kernel scale ρ .

For an objective computed from the normalized features in `compute_tau()`, take its derivative columns for location j and divide by $\sqrt{dJ}$ before passing them as `g_block`; J is the number of test locations.

Value

A list with two entries:

- `grad_vj`: $\nabla_{v_j} C$, a numeric vector of length d .
- `grad_param`: this location's contribution to $\partial C / \partial \rho$, one numeric value. Sum over locations when the scale is shared; zero for a kernel that exposes no squared scale.

References

Jitkrittum, W., Xu, W., Szabó, Z., Fukumizu, K., and Gretton, A. (2017). A Linear-Time Kernel Goodness-of-Fit Test. *Advances in Neural Information Processing Systems*, 30, 262–271. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2017/hash/979d472a84804b9f647bc185a877a8b5-Ab.html.

Examples

```
# Target: N(0, 1); d = J = 1.
X <- matrix(c(-1, 0, 1), ncol = 1)
kernel <- stein_kernel("gaussian_rbf", h = 1)
vj <- 1.5
tau <- compute_tau(X, -X, matrix(vj, nrow = 1), kernel)
sum(tau) # Objective C
g_block <- matrix(1, nrow(X), ncol(X)) # Derivative of sum: 1
fssd_grad_kernel(kernel, X, vj, grads_X = -X, g_block = g_block)
```

fssd_opt_test

FSSD test with optimized test locations

Description

Selects the test locations $L = \{v_1, \dots, v_J\}$ and, when the kernel exposes one, the squared kernel scale ρ on training rows. The FSSD statistic and null calibration use disjoint held-out rows.

Usage

```
fssd_opt_test(
  X,
  score_function,
  J = 5,
  n_simulations = 2000,
  kernel = c("gaussian_rbf", "imq"),
  scaling = NULL,
  train_ratio = 0.2,
  gamma = 1e-04,
  maxit = 100,
  locs_bounds_frac = 10,
  scale_lower = NULL,
```

```

    scale_upper = NULL,
    imq_beta = -0.5
)

```

Arguments

X	Numeric vector or matrix containing n independent and identically distributed observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix. At least four observations are required.
score_function	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
J	Number of test locations.
n_simulations	Number of null draws.
kernel	"gaussian_rbf", "imq", or a SteinKernel object.
scaling	Starting positive squared kernel scale (h^2 for Gaussian RBF, c^2 for IMQ). NULL uses a five-value grid around the training-row median squared scale when the kernel needs one. Supplying a value gives the starting value instead; an optimizable squared scale is still optimized from there. A finite squared scale in a supplied SteinKernel is the starting value and takes precedence; scaling is ignored when the kernel exposes no squared scale.
train_ratio	Requested FSSD-opt training fraction in $(0, 1)$; the random split keeps at least two rows in each part.
gamma	Positive regularizer in the FSSD-opt criterion.
maxit	Maximum L-BFGS-B iterations for FSSD-opt.
locs_bounds_frac	Number of training-set standard deviations added below each coordinate minimum and above each coordinate maximum.
scale_lower, scale_upper	Bounds applied to the squared kernel scale when that scale is optimized. NULL uses 0.01 and 100 times the median squared distance of the training rows, kept within $1e-3$ and $1e5$.
imq_beta	Finite IMQ exponent $\beta < 0$.

Details

The training criterion to maximize is

$$C(L, \rho) = \frac{\widehat{FSSD}_{\text{train}}^2}{\sqrt{\widehat{\text{Var}}_{H_1} + \gamma}},$$

where

$$\widehat{\text{Var}}_{H_1} = 4\bar{\tau}^T \widehat{\Sigma}_{\tau} \bar{\tau}, \quad \widehat{\Sigma}_{\tau} = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (\tau_i - \bar{\tau})(\tau_i - \bar{\tau})^T.$$

Here $\bar{\tau}$ is the mean feature vector on the training rows, and $\gamma > 0$ keeps the denominator away from zero.

Starting locations are drawn from a Gaussian fitted to the training rows. Bounded L-BFGS-B refines L and any optimizable squared scale jointly; the `scaling` argument specifies how its starting value is chosen. `fssd_grad_kernel()` supplies the gradient, one test location at a time.

The selected locations and kernel are fixed for the held-out test. For i.i.d. observations, choices based only on the training rows are independent of the held-out rows, as required by the null calibration in `fssd_null_pvalue()`. For the common interface and choice of variant, see `fssd_test()`.

Value

An object of class `hstest`. `statistic` is $S_{\tilde{n}} = \tilde{n} \widehat{FSSD}^2$, computed on the $\tilde{n}$ held-out rows; `p.value` is obtained from the held-out plug-in null calibration. `info` contains V (test locations), `n_train` and `n_test` (split sizes), `scale2_opt` (optimized squared scale), `objective_opt` (training criterion), `function_evaluations` (optimizer function evaluations), and `convergence` (0 for successful optimizer termination; otherwise see `stats::optim()`).

`scale_grid` and `scale_grid_objectives` record the optional initial scale search. Use `print(fit)` to display the test statistic and p-value.

References

Jitkrittum, W., Xu, W., Szabó, Z., Fukumizu, K., and Gretton, A. (2017). A Linear-Time Kernel Goodness-of-Fit Test. *Advances in Neural Information Processing Systems*, 30, 262–271. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2017/hash/979d472a84804b9f647bc185a877a8b5-Abstract.html.

Examples

```
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
score <- function(X) -X # Target: N(0, 1)
fit <- fssd_opt_test(X, score, J = 3, scaling = 1, train_ratio = 0.5)
c(statistic = unname(fit$statistic), p.value = fit$p.value)
fit$info$convergence # 0: successful optimizer termination.
```

fssd_rand_test

FSSD test with random test locations

Description

Fits a Gaussian distribution to X , draws the test locations $L = \{v_1, \dots, v_J\}$, and uses all rows of X to compute the FSSD statistic and its plug-in null approximation.

Usage

```
fssd_rand_test(
  X,
  score_function,
  J = 5,
```

```

    n_simulations = 2000,
    kernel = c("gaussian_rbf", "imq"),
    scaling = NULL,
    imq_beta = -0.5
  )

```

Arguments

<code>X</code>	Numeric vector or matrix containing n independent and identically distributed observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>J</code>	Number of test locations.
<code>n_simulations</code>	Number of null draws.
<code>kernel</code>	"gaussian_rbf", "imq", or a SteinKernel object.
<code>scaling</code>	Final positive squared kernel scale (h^2 for Gaussian RBF, c^2 for IMQ). NULL uses the square of the median of all pairwise distances in X when the kernel needs a squared scale. A Gaussian RBF kernel carrying precon uses its preconditioned distance; otherwise the distance is Euclidean. A finite squared scale in a supplied SteinKernel takes precedence; scaling is ignored when the kernel exposes no squared scale.
<code>imq_beta</code>	Finite IMQ exponent $\beta < 0$.

Details

Let $\bar{x}$ and $\hat{\Sigma}_X$ be the mean and fitted covariance of X . The locations are drawn independently as

$$v_j \sim N_d(\bar{x}, \hat{\Sigma}_X), \quad j = 1, \dots, J.$$

The same rows are used for the statistic and null simulation. The locations, and any data-derived kernel scale, use the tested rows, so the reported p-value is heuristic. See `fssd_null_pvalue()` for the independence condition. For the common interface and choice of variant, see `fssd_test()`.

Value

An object of class `hstest`. `statistic` is $S_n = n\widehat{FSSD}^2$; `p.value` is the same-sample plug-in p-value; and `info` contains the sampled $J \times d$ matrix V representing L . Use `print(fit)` to display the test statistic and p-value.

References

Jitkrittum, W., Xu, W., Szabó, Z., Fukumizu, K., and Gretton, A. (2017). A Linear-Time Kernel Goodness-of-Fit Test. *Advances in Neural Information Processing Systems*, 30, 262–271. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2017/hash/979d472a84804b9f647bc185a877a8b5-Abstract.html.

Examples

```
# Target: N(0, 1).
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
fit <- fssd_rand_test(X, function(X) -X, J = 3, scaling = 1)
fit$info$V
# Heuristic p-value
c(statistic = unname(fit$statistic), p.value = fit$p.value)
```

fssd_statistic

FSSD statistic and its null calibration

Description

Computes the scaled off-diagonal FSSD U-statistic from a feature matrix returned by [compute_tau\(\)](#), and performs the plug-in null calibration for that statistic. For the complete goodness-of-fit test, see [fssd_test\(\)](#).

Usage

```
fssd_statistic(tau_matrix)

fssd_null_pvalue(tau_matrix, statistic, n_simulations = 2000)
```

Arguments

tau_matrix Numeric $\tilde{n} \times dJ$ feature matrix returned by [compute_tau\(\)](#).

statistic Observed statistic $S_{\tilde{n}} = \tilde{n} \widehat{FSSD}^2$.

n_simulations Number of null draws to simulate.

Details

If τ_i is row i of τ_{matrix} and $\tilde{n}$ is its number of rows, the returned value is

$$S_{\tilde{n}} = \tilde{n} \widehat{FSSD}^2 = \frac{1}{\tilde{n} - 1} \sum_{i \neq j} \tau_i^T \tau_j.$$

Diagonal terms are excluded because population FSSD uses two independent draws. Although population $FSSD^2$ is nonnegative, its unbiased off-diagonal estimator and the returned $S_{\tilde{n}}$ can be negative in finite samples.

The fixed-location null calibration performed by [fssd_null_pvalue\(\)](#) requires the test locations L and the kernel, including its scale, to be fixed independently of the observations represented by τ_{matrix} , or selected using separate training data. That function cannot check that condition or correct same-sample selection.

Let $\tilde{n}$ be the number of rows of `tau_matrix`, and define

$$\widehat{\Sigma}_{\tau} = \frac{1}{\tilde{n}} \sum_{i=1}^{\tilde{n}} (\tau_i - \bar{\tau})(\tau_i - \bar{\tau})^T,$$

and let $\widehat{\lambda}_1, \dots, \widehat{\lambda}_{dJ}$ be its eigenvalues. Numerically negative eigenvalues are replaced by zero. For $b = 1, \dots, B$, the function simulates

$$S_{\tilde{n}}^{*(b)} = \sum_{q=1}^{dJ} \widehat{\lambda}_q \{(Z_q^{(b)})^2 - 1\}, \quad Z_q^{(b)} \sim N(0, 1),$$

and returns

$$\widehat{p} = \frac{1}{B} \sum_{b=1}^B \mathbf{1}\{S_{\tilde{n}}^{*(b)} \geq S_{\tilde{n}}\}.$$

The returned proportion does not use an add-one correction. The simulated law is the large- $\tilde{n}$ limit of $S_{\tilde{n}}$, so the test holds its nominal level only asymptotically.

Value

`fssd_statistic()` returns one numeric value: $S_{\tilde{n}} = \widehat{\tilde{n}FSSD}^2$. `fssd_null_pvalue()` returns a list with four entries:

- `p_value`: uncorrected simulated right-tail proportion.
- `statistic`: the supplied $S_{\tilde{n}}$.
- `null_samples`: simulated draws on exactly the same scale as `statistic`.
- `eigenvalues`: nonnegative eigenvalues of $\widehat{\Sigma}_{\tau}$ used in the null draws.

References

Jitkrittum, W., Xu, W., Szabó, Z., Fukumizu, K., and Gretton, A. (2017). A Linear-Time Kernel Goodness-of-Fit Test. *Advances in Neural Information Processing Systems*, 30, 262–271. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2017/hash/979d472a84804b9f647bc185a877a8b5-Abstract.html.

Examples

```
# Target: N(0, 1).
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
V <- matrix(c(-1, 0, 1), ncol = 1) # Prespecified test locations
kernel <- stein_kernel("gaussian_rbf", h = 1)
tau <- compute_tau(X, -X, V, kernel)
statistic <- fssd_statistic(tau)
result <- fssd_null_pvalue(tau, statistic)
c(statistic = result$statistic, p.value = result$p_value)
```

fssd_test

*Finite Set Stein Discrepancy goodness-of-fit test***Description**

Tests independent and identically distributed observations against a target distribution specified by its score function, using Stein features at a finite set of test locations. `variant = "opt"` selects those locations on training rows and tests on held-out rows (`fssd_opt_test()`); `variant = "rand"` draws them and tests on the same rows, so its p-value is heuristic (`fssd_rand_test()`).

Usage

```
fssd_test(
  X,
  score_function,
  variant = c("opt", "rand"),
  J = 5,
  n_simulations = 2000,
  kernel = c("gaussian_rbf", "imq"),
  scaling = NULL,
  train_ratio = 0.2,
  gamma = 1e-04,
  maxit = 100,
  locs_bounds_frac = 10,
  scale_lower = NULL,
  scale_upper = NULL,
  imq_beta = -0.5
)
```

Arguments

<code>X</code>	Numeric vector or matrix containing n independent and identically distributed observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>variant</code>	"opt" (default) or "rand".
<code>J</code>	Number of test locations.
<code>n_simulations</code>	Number of null draws.
<code>kernel</code>	"gaussian_rbf", "imq", or a SteinKernel object.
<code>scaling</code>	Positive squared kernel scale: h^2 for Gaussian RBF, c^2 for IMQ; a custom kernel may expose its own. It is final for FSSD-rand and an initial value for FSSD-opt. NULL uses a median-based value when the kernel needs a squared scale. A finite squared scale in a supplied SteinKernel takes precedence; scaling is ignored when the kernel exposes no squared scale.

train_ratio	Requested FSSD-opt training fraction in $(0, 1)$; the random split keeps at least two rows in each part.
gamma	Positive regularizer in the FSSD-opt criterion.
maxit	Maximum L-BFGS-B iterations for FSSD-opt.
locs_bounds_frac	Number of training-set standard deviations added below each coordinate minimum and above each coordinate maximum.
scale_lower, scale_upper	Bounds applied to the squared kernel scale when that scale is optimized. NULL uses 0.01 and 100 times the median squared distance of the training rows, kept within $1e-3$ and $1e5$.
imq_beta	Finite IMQ exponent $\beta < 0$.

Details

Once the test locations and kernel have been selected, `compute_tau()` evaluates the Stein features at those locations. `fssd_statistic()` combines the feature rows into the scaled test statistic, and `fssd_null_pvalue()` simulates its null distribution and computes the p-value.

Value

An `htest` object. `statistic` is $S_{\tilde{n}} = \tilde{n} \widehat{FSSD}^2$, where $\tilde{n}$ is the held-out size for FSSD-opt and n for FSSD-rand; `p.value` is its simulated right-tail p-value; `info$V` is the test-location matrix, and `info` contains variant-specific diagnostics. Use `print(fit)` to display the test statistic and p-value.

References

Jitkrittum, W., Xu, W., Szabó, Z., Fukumizu, K., and Gretton, A. (2017). A Linear-Time Kernel Goodness-of-Fit Test. *Advances in Neural Information Processing Systems*, 30, 262–271. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2017/hash/979d472a84804b9f647bc185a877a8b5-Abstract.html.

Examples

```
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
score <- function(X) -X # Target: N(0, 1)
fit <- fssd_test(X, score, J = 3, scaling = 1, train_ratio = 0.5)
c(statistic = unname(fit$statistic), p.value = fit$p.value)
fit$info$convergence # 0: successful optimizer termination.
```

<code>get_score_evaluator</code>	<i>Create a score function for a fixed Gaussian mixture model</i>
----------------------------------	---

Description

Returns a function(X) that evaluates the score of a `gmm()` object.

Usage

```
get_score_evaluator(model)
```

Arguments

`model` Gaussian mixture model returned by `gmm()`.

Details

Define the component responsibility

$$r_k(x) = \frac{w_k \phi(x; \mu_k, \Sigma_k)}{\sum_{\ell=1}^K w_\ell \phi(x; \mu_\ell, \Sigma_\ell)}.$$

The returned function evaluates

$$s_p(x) = \nabla_x \log p(x) = \sum_{k=1}^K r_k(x) \Sigma_k^{-1} (\mu_k - x).$$

Value

A function with signature `function(X)`. For an $n \times d$ matrix, it returns the corresponding $n \times d$ score matrix. For vector input, it returns a vector: multiple one-dimensional scores when `model$d == 1`, or one length- d score when `model$d > 1`.

Examples

```
model <- gmm(nComp = 1, mu = 0, sigma = 1)
score <- get_score_evaluator(model)
X <- matrix(c(-1, 0, 1), ncol = 1)
all.equal(score(X), -X)
```

gmm

*Create, sample, and evaluate a Gaussian mixture model***Description**

`gmm()` constructs a Gaussian mixture distribution with `nComp` components. `rgmm()` draws samples from it and `densitygmm()` evaluates its density.

Usage

```
gmm(nComp = NULL, mu = NULL, sigma = NULL, weights = NULL, d = NULL)
```

```
rgmm(model, n = 100)
```

```
densitygmm(model, X)
```

Arguments

<code>nComp</code>	Number of mixture components. If <code>NULL</code> , five components are generated.
<code>mu</code>	Component means, stored as a $d \times \text{nComp}$ matrix. A vector is also accepted for a one-dimensional mixture or a single component.
<code>sigma</code>	Component covariance matrices, stored as a $d \times d \times \text{nComp}$ array. Each matrix must be symmetric positive definite. A scalar or vector is treated as one-dimensional component variances; a $d \times d$ matrix is reused for every component.
<code>weights</code>	Optional nonnegative mixture weights, with at least one positive value. Values are normalized to sum to one.
<code>d</code>	Dimension of each sample. If omitted, it is inferred from <code>mu</code> or <code>sigma</code> , and otherwise defaults to one.
<code>model</code>	Gaussian mixture model returned by <code>gmm()</code> .
<code>n</code>	Number of samples to draw.
<code>X</code>	Finite numeric vector or matrix. For a one-dimensional model, a vector represents multiple observations. For a multivariate model, a length- <code>d</code> vector represents one observation. Matrix rows are observations.

Details

The model density is

$$p(x) = \sum_{k=1}^K w_k \phi(x; \mu_k, \Sigma_k),$$

where w_k , μ_k , and Σ_k are the weight, mean, and covariance of component k . Weights are normalized to sum to one. To use the model as a target, `get_score_evaluator()` supplies its score function. For each observation, `rgmm()` samples a component according to `model$weights`, then draws from that component's mean and covariance. `densitygmm()` evaluates this density at each observation and returns ordinary density values, not log densities.

With no arguments, `gmm()` creates a one-dimensional five-component mixture. If `mu` is omitted, its entries are drawn independently from $\text{Uniform}(0, 10)$ and centered across components within each coordinate. If `sigma` is omitted, every component uses the identity covariance matrix.

`summary(model)` lays out one row per mixture component with its weight, mean, and marginal standard deviations. `components` has one row per component: the weight w_k , the entries of μ_k , and $\sqrt{\text{diag}(\Sigma_k)}$. Off-diagonal covariance entries are not shown; read them from `model$sigma` when the components are correlated.

Value

`gmm()` returns an object of class "gmm" containing `nComp`; the $d \times \text{nComp}$ mean matrix `mu`; the $d \times d \times \text{nComp}$ covariance array `sigma`; normalized weights; and dimension `d`. `print(model)` displays the component count and dimension, and returns `model` invisibly. `summary(model)` returns a "summary.gmm" object with components `nComp`, `d`, and `components`. Printing that object returns it invisibly. `rgmm()` returns, for `model$d == 1`, a numeric vector of length `n`; for `model$d > 1`, an $n \times d$ numeric matrix with one observation per row. `densitygmm()` returns one density value per observation in `X`.

Examples

```
set.seed(1)
model <- gmm(nComp = 2, mu = cbind(c(-2, 0), c(2, 0)),
          sigma = array(diag(2), c(2, 2, 2)),
          weights = c(0.3, 0.7), d = 2)
summary(model)
X <- rgmm(model, n = 1000)
rbind(empirical = colMeans(X), target = drop(model$mu %*% model$weights))
head(densitygmm(model, X))
```

ksd_uq_matrix

Build the Stein-kernel matrix for the KSD tests

Description

Calculates the scores of the supplied observations and returns their Stein-kernel matrix.

Usage

```
ksd_uq_matrix(
  X,
  score_function,
  scaling = NULL,
  kernel = c("gaussian_rbf", "imq"),
  imq_beta = -0.5
)

ksd_vq_matrix(
  X,
```

```

    score_function,
    scaling = NULL,
    kernel = c("gaussian_rbf", "imq"),
    imq_beta = -0.5
)

```

Arguments

<code>X</code>	Numeric vector or matrix containing n observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>scaling</code>	Positive squared scale passed to the kernel unless a supplied <code>SteinKernel</code> already has one. For built-in RBF and IMQ kernels this is, respectively, h^2 and c^2 . For <code>kernel = "gaussian_rbf"</code> or <code>"imq"</code> , <code>NULL</code> uses the square of the median Euclidean distance between rows of <code>X</code> ; see <code>find_median_distance()</code> . For a supplied Gaussian RBF object without a fixed bandwidth, <code>NULL</code> instead uses the square of the median distance $\sqrt{(x-y)^\top M(x-y)}$, where M is <code>precon</code> , or the identity matrix when <code>precon = NULL</code> .
<code>kernel</code>	Kernel choice: <code>"gaussian_rbf"</code> , <code>"imq"</code> , or a <code>SteinKernel</code> object.
<code>imq_beta</code>	Finite exponent $\beta < 0$ of the built-in IMQ kernel.

Details

`ksd_uq_matrix()` and `ksd_vq_matrix()` are the same function under one name per test: the U- and V-statistics differ in how they use the matrix, not in how it is built.

For observations $X_1, \dots, X_n$, this function returns the full matrix

$$K_{ij} = k_{0,p}(X_i, X_j).$$

`ksd_u_statistic()` and `ksd_u_bootstrap()` exclude it; `ksd_v_statistic()` and `ksd_v_bootstrap()` retain it. For the complete tests, see `ksd_u_test()` and `ksd_v_test()`.

Value

Numeric $n \times n$ matrix K , including its diagonal.

References

- Liu, Q., Lee, J. D., and Jordan, M. I. (2016). A Kernelized Stein Discrepancy for Goodness-of-Fit Tests. Proceedings of the 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, 48, 276–284. <https://proceedings.mlr.press/v48/liub16.html>.
- Chwialkowski, K., Strathmann, H., and Gretton, A. (2016). A Kernel Test of Goodness of Fit. Proceedings of the 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, 48, 2606–2615. <https://proceedings.mlr.press/v48/chwialkowski16.html>.

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
score_function <- function(X) -X
ksd_uq_matrix(X, score_function)
```

ksd_u_statistic	<i>KSD-U statistic and its bootstrap</i>
-----------------	--

Description

Returns nU_n , calculated from the supplied Stein-kernel matrix with the diagonal excluded, and the bootstrap statistics used to calculate a KSD-U p-value.

Usage

```
ksd_u_statistic(K0)

ksd_u_bootstrap(
  K0,
  nboot = 1000,
  W_mat = NULL,
  boot_method = "multinomial_centered"
)
```

Arguments

K0	Finite numeric $n \times n$ Stein-kernel matrix with $n \geq 2$, such as the output of ksd_uq_matrix() .
nboot	Positive integer number of bootstrap draws. Ignored when W_mat is supplied.
W_mat	Optional finite numeric $n \times B$ multiplier matrix. If NULL, centered multinomial weights are generated. Supplied columns need not be centered.
boot_method	Bootstrap method. Currently only "multinomial_centered" is supported.

Details

For $K_{ij} = k_{0,p}(X_i, X_j)$, [ksd_u_statistic\(\)](#) returns

$$T_n = nU_n = \frac{1}{n-1} \sum_{i \neq j} K_{ij}.$$

The result can be negative.

For bootstrap draw b , let

$$w_i^{(b)} = \frac{N_i^{(b)}}{n} - \frac{1}{n}, \quad (N_1^{(b)}, \dots, N_n^{(b)}) \sim \text{Multinomial}\left(n; \frac{1}{n}, \dots, \frac{1}{n}\right).$$

The draw returned by `ksd_u_bootstrap()` is

$$T_n^{*(b)} = n \sum_{i \neq j} w_i^{(b)} w_j^{(b)} K_{ij}.$$

The draws are on the same scale as `ksd_u_statistic()`. Neither function computes a p-value. For the complete goodness-of-fit test, see `ksd_u_test()`.

Value

`ksd_u_statistic()` returns one numeric value, $T_n = nU_n$. `ksd_u_bootstrap()` returns a numeric vector containing $T_n^{*(1)}, \dots, T_n^{*(B)}$.

References

Liu, Q., Lee, J. D., and Jordan, M. I. (2016). A Kernelized Stein Discrepancy for Goodness-of-Fit Tests. Proceedings of the 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, 48, 276–284. <https://proceedings.mlr.press/v48/liub16.html>.

Examples

```
# Target: N(0, 1).
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
K0 <- ksd_uq_matrix(X, function(X) -X, scaling = 1)
statistic <- ksd_u_statistic(K0)
draws <- ksd_u_bootstrap(K0)
c(statistic = statistic,
  p.value = (1 + sum(draws >= statistic)) / (length(draws) + 1))
```

ksd_u_test

KSD-U goodness-of-fit test for independent observations

Description

Tests whether independent observations come from a target distribution specified by its score function. The target density need not be normalized.

Usage

```
ksd_u_test(
  X,
  score_function,
  boot_method = "multinomial_centered",
  scaling = NULL,
  nboot = 1000,
  kernel = c("gaussian_rbf", "imq"),
  return_raw_boot = FALSE,
```

```

    block_size = NULL,
    block_threshold = 5000,
    imq_beta = -0.5
)

```

Arguments

<code>X</code>	Numeric vector or matrix containing n observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>boot_method</code>	Bootstrap method. Currently only "multinomial_centered" is supported.
<code>scaling</code>	Positive squared scale passed to the kernel unless a supplied <code>SteinKernel</code> already has one. For built-in RBF and IMQ kernels this is, respectively, h^2 and c^2 . For kernel = "gaussian_rbf" or "imq", NULL uses the square of the median Euclidean distance between rows of <code>X</code> ; see <code>find_median_distance()</code> . For a supplied Gaussian RBF object without a fixed bandwidth, NULL instead uses the square of the median distance $\sqrt{(x-y)^\top M(x-y)}$, where M is precon, or the identity matrix when precon = NULL.
<code>nboot</code>	Positive integer number of bootstrap draws B .
<code>kernel</code>	Kernel choice: "gaussian_rbf", "imq", or a <code>SteinKernel</code> object.
<code>return_raw_boot</code>	Logical. If TRUE, include the B bootstrap draws in the result.
<code>block_size</code>	Positive integer number of rows per block. NULL uses $\min(1024, n)$ when $n > \text{block_threshold}$ and n otherwise. Blocking changes memory use, not the test definition.
<code>block_threshold</code>	Positive integer sample-size threshold above which blockwise computation is used.
<code>imq_beta</code>	Finite exponent $\beta < 0$ of the built-in IMQ kernel.

Details

Let $s_p(x) = \nabla_x \log p(x)$ and

$$K_{ij} = k_{0,p}(X_i, X_j),$$

where $k_{0,p}$ is the Stein kernel defined in `stein_kernel_matrix()`. To construct this matrix separately from the observations, score function, and kernel, use `ksd_uq_matrix()`. The test uses

$$U_n = \frac{1}{n(n-1)} \sum_{i \neq j} K_{ij}, \quad T_n = nU_n.$$

The diagonal is excluded. Thus U_n is unbiased for the population squared KSD, but its finite-sample value can be negative. The scaled statistic T_n can be computed separately with `ksd_u_statistic()`.

Null calibration uses the centered multinomial bootstrap implemented by `ksd_u_bootstrap()`. If $T_n^{*(1)}, \dots, T_n^{*(B)}$ are the bootstrap draws, the reported right-tail p-value is

$$\hat{p} = \frac{1 + \sum_{b=1}^B \mathbf{1}\{T_n^{*(b)} \geq T_n\}}{B + 1}.$$

Warns when off-diagonal Stein-kernel magnitudes are negligible relative to the diagonal, because the resulting bootstrap calibration is degenerate.

Value

An `htest` object. `statistic` is $T_n = nU_n$; `p.value` is the bootstrap p-value above. `parameter` records `nboot` and `scaling`, plus `imq_beta` for an IMQ kernel; `kernel` contains the resolved SteinKernel object. If `return_raw_boot = TRUE`, `bootstrap_samples` contains $T_n^{*(1)}, \dots, T_n^{*(B)}$. Use `print(fit)` to display the test statistic and p-value.

References

Liu, Q., Lee, J. D., and Jordan, M. I. (2016). A Kernelized Stein Discrepancy for Goodness-of-Fit Tests. Proceedings of the 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, 48, 276–284. <https://proceedings.mlr.press/v48/liub16.html>.

Examples

```
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
score <- function(X) -X # Target: N(0, 1)
fit <- ksd_u_test(X, score, scaling = 1)
print(fit)
```

<code>ksd_v_statistic</code>	<i>KSD-V statistic and its bootstrap</i>
------------------------------	--

Description

Returns nV_n , calculated from the supplied Stein-kernel matrix with the diagonal included, and the bootstrap statistics used to calculate a KSD-V p-value.

Usage

```
ksd_v_statistic(K0)

ksd_v_bootstrap(
  K0,
  nboot = 1000,
  W_mat = NULL,
  boot_method = c("rademacher", "markov"),
  change_prob = NULL
)
```

Arguments

<code>K0</code>	Finite numeric $n \times n$ Stein-kernel matrix with $n \geq 2$, such as the output of <code>ksd_vq_matrix()</code> .
<code>nboot</code>	Positive integer number of bootstrap draws. Ignored when <code>W_mat</code> is supplied.
<code>W_mat</code>	Optional finite numeric $n \times B$ multiplier matrix. If NULL, signs are generated from <code>boot_method</code> . Supplied entries need not be ± 1 .
<code>boot_method</code>	Sign process used when <code>W_mat</code> = NULL: "rademacher" or "markov".
<code>change_prob</code>	Markov sign-change probability a_n . It must be supplied and lie strictly between 0 and 1 when <code>boot_method</code> = "markov".

Details

For $K_{ij} = k_{0,p}(X_i, X_j)$, `ksd_v_statistic()` returns

$$nV_n = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^n K_{ij}.$$

For sign vector $W^{(b)}$, the draw returned by `ksd_v_bootstrap()` is

$$nV_n^{*(b)} = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^n W_i^{(b)} W_j^{(b)} K_{ij}.$$

Rademacher signs are independent and take values -1 and 1 with equal probability. Markov signs satisfy

$$W_1^{(b)} = 1, \quad W_t^{(b)} = \begin{cases} -W_{t-1}^{(b)}, & \text{with probability } a_n, \\ W_{t-1}^{(b)}, & \text{with probability } 1 - a_n. \end{cases}$$

Here a_n is `change_prob`. The draws are on the same scale as `ksd_v_statistic()`. Neither function computes a p-value. For the complete goodness-of-fit test, see `ksd_v_test()`.

Value

`ksd_v_statistic()` returns one numeric value, nV_n . `ksd_v_bootstrap()` returns a numeric vector containing $nV_n^{*(1)}, \dots, nV_n^{*(B)}$.

References

Chwialkowski, K., Strathmann, H., and Gretton, A. (2016). A Kernel Test of Goodness of Fit. Proceedings of the 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, 48, 2606–2615. <https://proceedings.mlr.press/v48/chwialkowski16.html>.

Examples

```
# Target: N(0, 1).
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
K0 <- ksd_vq_matrix(X, function(X) -X, scaling = 1)
statistic <- ksd_v_statistic(K0)
draws <- ksd_v_bootstrap(K0)
c(statistic = statistic,
  p.value = (1 + sum(draws >= statistic)) / (length(draws) + 1))
```

ksd_v_test

KSD-V goodness-of-fit test with wild-bootstrap calibration

Description

Tests observations against a target distribution specified by its score function. Use Rademacher calibration for independent observations and Markov calibration for ordered dependent observations.

Usage

```
ksd_v_test(
  X,
  score_function,
  boot_method = c("rademacher", "markov"),
  scaling = NULL,
  nboot = 1000,
  change_prob = NULL,
  kernel = c("gaussian_rbf", "imq"),
  return_raw_boot = FALSE,
  block_size = NULL,
  block_threshold = 5000,
  imq_beta = -0.5
)
```

Arguments

<code>X</code>	Numeric vector or matrix containing n observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix. For Markov calibration, rows must follow the dependence order.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>boot_method</code>	Calibration method: "rademacher" for independent observations or "markov" for ordered dependent observations.
<code>scaling</code>	Positive squared scale passed to the kernel unless a supplied SteinKernel already has one. For built-in RBF and IMQ kernels this is, respectively, h^2 and c^2 . For kernel = "gaussian_rbf" or "imq", NULL uses the square of the median Euclidean distance between rows of X; see find_median_distance() . For

	a supplied Gaussian RBF object without a fixed bandwidth, NULL instead uses the square of the median distance $\sqrt{(x-y)^\top M(x-y)}$, where M is precon, or the identity matrix when precon = NULL.
nboot	Positive integer number of bootstrap draws B .
change_prob	Sign-change probability a_n for Markov calibration. It must be supplied and lie strictly between 0 and 1. It is ignored for Rademacher calibration.
kernel	Kernel choice: "gaussian_rbf", "imq", or a SteinKernel object.
return_raw_boot	Logical. If TRUE, include the B bootstrap draws in the result.
block_size	Positive integer number of rows per block. NULL uses $\min(1024, n)$ when $n > \text{block_threshold}$ and n otherwise. Blocking changes memory use, not the test definition.
block_threshold	Positive integer sample-size threshold above which blockwise computation is used.
imq_beta	Finite exponent $\beta < 0$ of the built-in IMQ kernel.

Details

Let $s_p(x) = \nabla_x \log p(x)$ and

$$K_{ij} = k_{0,p}(X_i, X_j),$$

where $k_{0,p}$ is defined in [stein_kernel_matrix\(\)](#). To construct this matrix separately from the observations, score function, and kernel, use [ksd_vq_matrix\(\)](#). The test uses

$$V_n = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n K_{ij}, \quad nV_n = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^n K_{ij}.$$

The diagonal is retained. For a positive-definite base kernel, V_n is nonnegative but upward biased for the population squared KSD. The scaled statistic nV_n can be computed separately with [ksd_v_statistic\(\)](#).

`boot_method = "rademacher"` uses independent signs and is intended for independent observations. `boot_method = "markov"` uses correlated signs and requires rows of X to be in dependence order. The latter calibration is valid only under the dependence, moment, and kernel assumptions of Chwialkowski et al. (2016); this function does not check them. The sign-change probabilities a_n must asymptotically satisfy $a_n \rightarrow 0$ and $na_n \rightarrow \infty$.

[ksd_v_bootstrap\(\)](#) generates bootstrap draws on the same scale as the observed statistic. If $nV_n^{*(1)}, \dots, nV_n^{*(B)}$ are these draws, the reported right-tail p-value is

$$\hat{p} = \frac{1 + \sum_{b=1}^B \mathbf{1}\{nV_n^{*(b)} \geq nV_n\}}{B + 1}.$$

Warns when off-diagonal Stein-kernel magnitudes are negligible relative to the diagonal, because the resulting bootstrap calibration is degenerate.

Value

An `hstest` object. `statistic` is nV_n ; `p.value` is the bootstrap p-value above. `parameter` records `nboot`, `scaling`, and `change_prob`, plus `imq_beta` for an IMQ kernel; `kernel` contains the resolved `SteinKernel` object.

With `return_raw_boot = TRUE`, `bootstrap_samples` holds $nV_n^{*(1)}, \dots, nV_n^{*(B)}$. Use `print(fit)` to display the test statistic and p-value.

References

Chwialkowski, K., Strathmann, H., and Gretton, A. (2016). A Kernel Test of Goodness of Fit. Proceedings of the 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, 48, 2606–2615. <https://proceedings.mlr.press/v48/chwialkowski16.html>.

Examples

```
set.seed(1)
X <- matrix(rnorm(200, mean = 0.5), ncol = 1)
score <- function(X) -X # Target: N(0, 1)
fit <- ksd_v_test(X, score, scaling = 1)
print(fit)
```

mala

Markov transitions for SP-MCMC

Description

Runs the MALA or the Gaussian random-walk Metropolis transition used by `sp_mcmc()`.

Usage

```
mala(log_p, score_function, x0, h, Sigma = NULL, m_iter)
```

```
rwm(log_p, x0, h, Sigma = NULL, m_iter)
```

Arguments

<code>log_p</code>	Function taking an $n \times d$ matrix and returning n log-density values.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>x0</code>	Finite initial state vector.
<code>h</code>	Positive step-size multiplier; the proposal covariance is $h * \text{Sigma}$.
<code>Sigma</code>	Symmetric positive-definite proposal covariance scale. For <code>mala()</code> it preconditions both the drift and the noise. If <code>NULL</code> , the identity matrix is used.
<code>m_iter</code>	Number of returned chain rows, including <code>x0</code> in row 1.

Details

From the current state x , the `mala()` proposal is

$$y = x + (h/2)\Sigma s_p(x) + z, \quad z \sim N(0, h\Sigma),$$

so the proposal covariance is $h * \Sigma$. The function evaluates $\log_p$ at y ; $-\text{Inf}$ rejects the proposal. Otherwise it evaluates the score at y , forms the reverse proposal, and applies the Metropolis correction. Row 1 of the output is x_0 ; $m_iter - 1$ proposals follow.

From the current state x , the `rwm()` proposal is

$$y = x + z, \quad z \sim N(0, h\Sigma).$$

The function evaluates $\log_p(y)$ and accepts with probability $\min(1, \exp(\log_p(y) - \log_p(x)))$; $-\text{Inf}$ is rejected. Row 1 of the output is x_0 ; $m_iter - 1$ proposals follow. RWM does not evaluate the score. `sp_mcmc_eval_candidates()` computes candidate scores later.

Value

`mala()` returns a list with:

- X : chain states.
- D : scores at those states, reused by SP-MCMC.
- $\log_p$: log-density values at those states.
- accept : 0/1 indicators, with 0 in row 1.
- counts : numbers of rows evaluated by $\log_p$ and score_function ; $n_eval = \text{counts}\$total$.

`rwm()` returns a list with:

- X : chain states.
- $\log_p$: log-density values at those states.
- accept : 0/1 indicators, with 0 in row 1.
- $D = \text{NULL}$, because RWM computes no scores.
- counts : row counts with $\text{counts}\$score = 0$; $n_eval = \text{counts}\$total = \text{counts}\$\log_p$.

References

Chen, W. Y., Barp, A., Briol, F.-X., Gorham, J., Girolami, M., Mackey, L., and Oates, C. J. (2019). Stein Point Markov Chain Monte Carlo. Proceedings of the 36th International Conference on Machine Learning, Proceedings of Machine Learning Research, 97, 1011–1021. <https://proceedings.mlr.press/v97/chen19b.html>.

Examples

```
# Target: N(0, 1).
set.seed(1)
score <- function(X) -X
log_p <- function(X) -0.5 * rowSums(X^2)
x0 <- rnorm(1)
```

```
chains <- list(mala = mala(log_p, score, x0, h = 1, m_iter = 1000),
              rwm = rwm(log_p, x0, h = 1, m_iter = 1000))
# Acceptance rate over proposals (rows 2:1000).
vapply(chains, function(z) mean(z$accept[-1]), numeric(1))
```

sp_mcmc

*Select Stein points from short Markov chains***Description**

Builds a point set sequentially. At each step, a short Markov chain supplies candidates, and the state that adds the least to the Stein-kernel sum is kept.

Usage

```
sp_mcmc(
  score_function,
  log_p,
  kernel,
  n_points,
  d,
  mcmc = c("rwm", "mala"),
  criterion = c("last", "rand", "infl"),
  m_seq,
  h,
  Sigma = NULL,
  x_init,
  transition_fn = NULL,
  proposal_fn = NULL
)

sp_mcmc_eval_candidates(
  kernel,
  score_function,
  X_curr,
  D_curr,
  cand_X,
  cand_D = NULL
)
```

Arguments

score_function Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores. [sp_mcmc_eval_candidates\(\)](#) uses it to score candidate rows.

log_p Function taking an $n \times d$ matrix and returning n log-density values.

kernel A SteinKernel object. A Gaussian RBF kernel must have a fixed positive bandwidth.

n_points	Total number of points, including x_init.
d	State dimension.
mcmc	MCMC transition: "rwm" for Gaussian random-walk Metropolis or "mala" for MALA.
criterion	Path-start rule. "last" uses the newest point, "rand" samples uniformly, and "infl" uses the point whose removal maximizes the remaining-set KSD. A custom rule is a list containing select and an optional label.
m_seq	Number of path states before duplicate removal. A scalar is reused; a vector of length n_points - 1 configures each added point.
h	Positive multiplier in the proposal covariance h * Sigma.
Sigma	Symmetric positive-definite matrix used by the built-in transitions. If NULL, the identity matrix is used.
x_init	Initial finite numeric vector of length d.
transition_fn	Optional function replacing <code>rwm()</code> or <code>mala()</code> .
proposal_fn	Optional per-step update of h or Sigma; see Extensions.
X_curr	Current selected point matrix.
D_curr	Current score matrix.
cand_X	Candidate point matrix.
cand_D	Optional candidate score matrix.

Details

At step j , the function generates a path, removes repeated rows, and appends the row minimizing the greedy objective

$$G_j(x) = k_{0,p}(x, x) + 2 \sum_{i=1}^{j-1} k_{0,p}(x_i, x),$$

where $k_{0,p}$ is the Stein kernel formed from kernel and the target score. The returned ksd is the running discrepancy defined in `stein_points()`. `sp_mcmc_eval_candidates()` evaluates the greedy objective for the candidate rows, reusing their scores when available.

criterion chooses the selected point where each path starts. A path with m_seq states makes m_seq - 1 transitions. Built-in paths use `rwm()` or `mala()` with proposal covariance h * Sigma; MALA also uses Sigma in its drift. To use a covariance S for both the path and kernel distance, set Sigma = S and the kernel's precon = solve(S).

summary(fit) reports accept_rate weighted by the number of transitions in each path; paths without acceptance diagnostics are omitted. n_repeated counts selected points that repeat an earlier one; a candidate path contains its own starting state, so repeats are expected. The reported h is the initial step size, before any per-step proposal changes. counts contains the log_p, score, and candidate_score column totals; n_eval_total reports their total once.

Value

An "sp_mcmc" object with:

- X and D: $n_{\text{points}} \times d$ matrices of selected points and their scores.

- ksd: running discrepancy after each point.
- n_eval: evaluations at each step; `cum_n_eval = cumsum(n_eval)`.
- counts: the `log_p`, `score`, and `candidate_score` counts at each step.
- selected_index: row of X that each path started from.
- accept_rate: share of accepted proposals in each path.
- chain_d2_max, chain_d2_selected, chain_d2_last: squared distances from each path's starting state to its farthest state, the state kept, and its last state.
- n_repeated: number of selected points that repeat an earlier one.
- mcmc, transition, criterion, m_seq, h, Sigma, and kernel: settings used by the run.
- method and call: method label and matched call.

The first point has no path, so its diagnostic entries are NA. Recorded `h` and `Sigma` are the original inputs, not step-specific replacements.

`sp_mcmc_eval_candidates()` returns a list with:

- objective_values: one comparison value per candidate row.
- scores: candidate scores, reused from `cand_D` when supplied.
- k0_diag: Stein-kernel diagonal for the candidate rows.
- score_evaluations: rows scored here; zero when `cand_D` is supplied.

`print(fit)` displays the transition method, selection rule, point-set size, final KSD, and evaluation count, and returns `fit` invisibly. `summary(fit)` returns a `"summary.sp_mcmc"` object with `transition`, `criterion`, `m_seq`, `h`, `accept_rate`, `n_repeated`, and `counts`. Printing that object returns it invisibly.

Extensions

Supply a custom start rule as `list(select = function(state) ..., label = "custom")`. `select` returns one integer index into `state$X`; `label` defaults to `"custom"`. `state` contains `j`, `X`, `D`, `K0`, recorded diagnostics, and run settings.

A custom `transition_fn(log_p, score_function, x0, h, Sigma, m_iter)` returns `X` and `counts`; `X` starts at `x0`, optional `D` holds the scores of `X`, and optional `accept` is a length-`m_iter` vector of 0/1 indicators with initial entry 0. `counts` contains nonnegative integer `log_p` and `score` counts.

`proposal_fn(j, X_curr, h, Sigma, mcmc)` returns a list containing `h` and/or `Sigma`; an empty list keeps both inputs. Changes apply only at step `j`.

`sp_mcmc_eval_candidates()` returns one greedy-objective value per row of `cand_X`. Supply `cand_D` to reuse candidate scores; otherwise `score_function` evaluates them.

References

Chen, W. Y., Barp, A., Briol, F.-X., Gorham, J., Girolami, M., Mackey, L., and Oates, C. J. (2019). Stein Point Markov Chain Monte Carlo. Proceedings of the 36th International Conference on Machine Learning, Proceedings of Machine Learning Research, 97, 1011–1021. <https://proceedings.mlr.press/v97/chen19b.html>.

See Also

[stein_points\(\)](#), [rwm\(\)](#), [mala\(\)](#)

Examples

```
# Target: N(0, 1).
set.seed(1)
score <- function(X) -X
log_p <- function(X) -0.5 * rowSums(X^2)
kernel <- stein_kernel("imq")
fit <- sp_mcmc(score, log_p, kernel, n_points = 50, d = 1,
              m_seq = 50, h = 1, x_init = 0)
summary(fit)
c(mean = mean(fit$X), second_moment = mean(fit$X^2)) # Targets: 0 and 1

X_curr <- matrix(0, ncol = 1)
D_curr <- score(X_curr)
cand_X <- matrix(c(-0.5, 0.5), ncol = 1)
sp_mcmc_eval_candidates(kernel, score, X_curr, D_curr, cand_X)
```

stein_codescent

Refine Stein Points by coordinate descent

Description

Refines a completed Stein point set one row at a time while holding the others fixed. A replacement is accepted only if the point-set KSD does not increase.

Usage

```
stein_codescent(X0, score_function, kernel, n_iter, optimizer)
```

Arguments

<code>X0</code>	Initial $n \times d$ point matrix; rows are points.
<code>score_function</code>	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
<code>kernel</code>	A <code>SteinKernel</code> object.
<code>n_iter</code>	Number of coordinate-descent updates.
<code>optimizer</code>	Optimizer function used for each coordinate update, such as fmin_grid() , fmin_mc() , or fmin_nm() .

Details

At iteration it , row $r = ((it - 1) \% nrow(X0)) + 1$ is replaced by minimizing

$$k_{0,p}(x, x) + 2 \sum_{i \neq r} k_{0,p}(x_i, x)$$

over x , where $k_{0,p}$ is the Stein kernel formed from `kernel` and the target score. This is the greedy objective of `stein_points()`, except the set size stays fixed. With a single row the sum is empty and the objective is $k_{0,p}(x, x)$.

Because the optimizers are approximate, each proposal is compared with the current row under the same objective before it is accepted. The optimizer interface is the same as in `stein_points()`.

Value

An object of class "stein_codescent" with:

- `X`: refined point matrix with the same dimensions as `X0`.
- `D`: target scores at the refined points.
- `objective`: retained value of the coordinate objective at each update. These are not KSDs and are not comparable across updates, because the objective omits the kernel sum over the fixed rows, which changes as the points move.
- `n_eval`: optimizer-reported evaluations per update; the first entry also counts the `nrow(X0)` initial scores.
- `cum_n_eval = cumsum(n_eval)`.
- `kernel`: kernel used to define the Stein objective.

No running ksd is returned. Compute the final KSD as `sqrt(sum(stein_kernel_matrix(kernel, out$X, out$D))) / nrow(out$X)`.

`print(fit)` displays the point-set size, update count, and evaluation count, and returns `fit` invisibly. `summary(fit)` returns a "summary.stein_codescent" object with components `method`, `kernel`, `n`, `d`, `evaluation_label`, `n_eval_total`, `call`, and `n_iter`; `evaluation_label` names what `n_eval_total` counts. Coordinate-descent objectives are not displayed because they change with the row being updated and are not point-set KSDs. Printing that object returns it invisibly.

References

Chen, W. Y., Mackey, L., Gorham, J., Briol, F.-X., and Oates, C. J. (2018). Stein Points. Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 80, 844–853. <https://proceedings.mlr.press/v80/chen18f.html>.

Examples

```
# Target: N(0, 1).
score <- function(X) -X
kernel <- stein_kernel("imq")
X0 <- matrix(seq(-0.5, 0.5, length.out = 20), ncol = 1)
opt <- fmin_grid(-4, 4, n0 = 401, grow = FALSE)
fit <- stein_codescent(X0, score, kernel, n_iter = 40, optimizer = opt)
```

```
ksd <- function(X) {
  sqrt(max(0, sum(stein_kernel_matrix(kernel, X, score(X))))) / nrow(X)
}
c(before = ksd(X0), after = ksd(fit$X))
```

stein_kernel	Create a built-in Stein kernel
--------------	--------------------------------

Description

Creates a Gaussian RBF or IMQ kernel with the chosen parameters.

Usage

```
stein_kernel(
  type = c("gaussian_rbf", "imq"),
  h = NULL,
  beta = -0.5,
  c = 1,
  precon = NULL
)
```

Arguments

type	"gaussian_rbf" or "imq".
h	Gaussian RBF bandwidth $h > 0$. If NULL, the kernel has no fixed bandwidth.
beta	Finite IMQ exponent $\beta < 0$.
c	Positive IMQ length scale c .
precon	Optional symmetric positive-definite M .

Details

With $r_M(x, y) = (x - y)^\top M(x - y)$ and $M = \text{precon}$ (identity when $\text{precon} = \text{NULL}$), the base kernels are

$$k_{\text{RBF}}(x, y) = \exp\{-r_M(x, y)/(2h^2)\}, \quad k_{\text{IMQ}}(x, y) = \{c^2 + r_M(x, y)\}^\beta.$$

[scale2_kernel\(\)](#) reads or replaces the squared scale h^2 or c^2 . For a different kernel formula, [custom_stein_kernel\(\)](#) accepts user-supplied calculations.

Two further kernels, [stein_kernel_inverse_log\(\)](#) and [stein_kernel_imq_score\(\)](#), are provided for [stein_points\(\)](#) and [sp_mcmc\(\)](#).

Value

[stein_kernel\(\)](#) returns a `SteinKernel` object. [print\(\)](#) displays the kernel type, parameters, and the dimensions of the preconditioning matrix, when present, and returns its argument invisibly.

Examples

```
kernel <- stein_kernel("gaussian_rbf", h = 2)
print(kernel)
stein_kernel("imq", c = 2, beta = -0.25)
```

stein_kernel_imq_score

Create a score-distance IMQ Stein kernel

Description

Constructs an IMQ kernel from distances between target-score vectors.

Usage

```
stein_kernel_imq_score(alpha = 1, beta = -0.5, hess_log_p)
```

Arguments

alpha	Positive offset parameter.
beta	Exponent in $(-1, 0)$.
hess_log_p	Function returning an $n \times d \times d$ Hessian array.

Details

With $s_p(x) = \nabla_x \log p(x)$, the base kernel is

$$k(x, y) = (\alpha + \|s_p(x) - s_p(y)\|^2)^\beta.$$

Its Stein derivatives require `hess_log_p`, which returns an $n \times d \times d$ array containing one Hessian of the target log density per row of `X`. Preconditioning is not supported.

Use `stein_kernel_matrix()` to evaluate this kernel; `eval_kernel()`, `grad_x_kernel()`, `trace_mixed_kernel()`, and `cross_kernel()` are not supported.

Value

A `SteinKernel` object for the score-distance IMQ kernel. Use `print(kernel)` to display the kernel type and parameters.

References

Chen, W. Y., Mackey, L., Gorham, J., Briol, F.-X., and Oates, C. J. (2018). Stein Points. Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 80, 844–853. <https://proceedings.mlr.press/v80/chen18f.html>.

See Also

`stein_kernel()` for the other built-in kernel choices.

Examples

```
hess_log_p <- function(X) array(-1, dim = c(nrow(as.matrix(X)), 1, 1))
kernel <- stein_kernel_imq_score(alpha = 1.5, beta = -0.25,
                                hess_log_p = hess_log_p)
print(kernel)
```

stein_kernel_inverse_log

Create an inverse-log Stein kernel

Description

Constructs a kernel from the logarithm of the squared distance between points.

Usage

```
stein_kernel_inverse_log(alpha = 1, beta = -1)
```

Arguments

alpha	Positive offset parameter.
beta	Negative exponent.

Details

The base kernel has the form

$$k(x, y) = (\alpha + \log(1 + \|x - y\|^2))^\beta.$$

The parameter alpha must be positive and beta must be negative; the default $\beta = -1$ gives the plain inverse-log kernel. Compared with a Gaussian RBF kernel, this kernel decays much more slowly as points move apart and retains interactions between distant points.

This kernel uses Euclidean distance and does not support preconditioning.

Value

A SteinKernel object for the inverse-log kernel. Use `print(kernel)` to display the kernel type and parameters.

References

Chen, W. Y., Mackey, L., Gorham, J., Briol, F.-X., and Oates, C. J. (2018). Stein Points. Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 80, 844–853. <https://proceedings.mlr.press/v80/chen18f.html>.

See Also

[stein_kernel\(\)](#) for the other built-in kernel choices.

Examples

```
kernel <- stein_kernel_inverse_log(alpha = 2, beta = -0.5)
print(kernel)
```

stein_kernel_matrix	<i>Assemble the pairwise Stein-kernel matrix</i>
---------------------	--

Description

Returns the Stein-kernel matrix K , with entries $K_{ij} = k_{0,p}(x_i, y_j)$.

Usage

```
stein_kernel_matrix(kernel, X, grads, Y = NULL, grads_Y = NULL, precon = NULL)
```

Arguments

kernel	A SteinKernel object.
X, Y	Numeric point matrices; Y = NULL uses X.
grads, grads_Y	Score matrices matching X and Y.
precon	Optional preconditioner overriding kernel\$precon.

Details

With $s_p(x) = \nabla_x \log p(x)$,

$$k_{0,p}(x, y) = s_p(x)^\top s_p(y) k(x, y) + s_p(x)^\top \nabla_y k(x, y) + s_p(y)^\top \nabla_x k(x, y) + \text{tr}\{\nabla_x \nabla_y^\top k(x, y)\}.$$

[eval_kernel\(\)](#) supplies the base-kernel values, [cross_kernel\(\)](#) the two score-derivative terms, and [trace_mixed_kernel\(\)](#) the mixed-derivative trace. For kernel construction, see [stein_kernel\(\)](#).

Value

Numeric $n_X \times n_Y$ matrix K .

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
stein_kernel_matrix(stein_kernel("gaussian_rbf", h = 1), X, -X)
```

stein_points	<i>Construct points by Stein discrepancy minimization</i>
--------------	---

Description

Builds a point set sequentially. At each step, optimizer searches the continuous state space for the next point under a greedy or herding Stein objective.

Usage

```
stein_points(
    score_function,
    kernel,
    n_points,
    d,
    optimizer,
    method = c("greedy", "herding"),
    log_p = NULL,
    x_init = NULL,
    c2 = NULL,
    truncation = c("none", "upper", "lower", "linear")
)
```

Arguments

score_function	Function taking an $n \times d$ matrix and returning the corresponding $n \times d$ matrix of target scores.
kernel	A SteinKernel object used to construct $k_{0,p}$. A Gaussian RBF kernel must use a fixed positive bandwidth.
n_points	Positive number of points to select.
d	Positive state dimension.
optimizer	Function taking (objective, X_curr, t) and returning x_min and its score d_min as length-d vectors and a nonnegative integer n_eval. objective(X_new, scores = NULL) is vectorized over the rows of X_new and returns a list: objective_values holds one value per candidate row, the value to minimize, and scores holds the scores it used, which supply d_min without scoring the selected point again. X_curr holds the points kept fixed at this step, and t is the step counter. The returned vectors must be finite. During initialization from log_p, only x_min and n_eval are used; the selected point is scored separately. With truncation, the optimizer must select a feasible candidate; the selected point is checked before it is appended. An infeasible result raises an error. The selected point's objective is recomputed before updating the running KSD.
method	Point-selection rule: "greedy" or "herding".
log_p	Log-density function taking an $n \times d$ matrix and returning n values; required when x_init is NULL, otherwise ignored. The density need not be normalized.

x_init	Optional finite numeric vector of length d giving the first point.
c2	Finite positive denominator in the truncation radius. Ignored when truncation = "none".
truncation	Candidate filter described under Details: "none", "upper", "lower", or "linear".

Details

Let $k_{0,p}$ be the Stein kernel formed from kernel and $s_p(x) = \nabla_x \log p(x)$. The first point is x_init; if x_init is NULL, optimizer instead maximizes log_p.

Given selected points $x_1, \dots, x_{j-1}$, the greedy optimizer minimizes

$$G_j(x) = k_{0,p}(x, x) + 2 \sum_{i=1}^{j-1} k_{0,p}(x_i, x).$$

This is the amount added to the kernel sum used by the running KSD. With method = "herding", the optimizer instead minimizes

$$H_j(x) = \sum_{i=1}^{j-1} k_{0,p}(x_i, x).$$

The supplied `fmin_grid()`, `fmin_mc()`, and `fmin_nm()` constructors use grid, Monte Carlo, and Nelder-Mead search, respectively. Supply a custom optimizer with the interface described under that argument to replace this search. To refine a completed point set, use `stein_codescent()`.

The returned discrepancy after step j is

$$\text{KSD}_j = \left\{ \frac{1}{j^2} \sum_{a=1}^j \sum_{b=1}^j k_{0,p}(x_a, x_b) \right\}^{1/2}.$$

Truncation keeps candidates satisfying

$$k_{0,p}(x, x) \leq R_j^2.$$

Write $U^2 = 2 \log\{\max(n_{\text{points}}, 2)\}/c2$ and $L_j^2 = 2 \log(j)/c2$. "upper" uses U^2 , "lower" uses L_j^2 , and "linear" uses $L_j^2 + (U^2 - L_j^2)(j - 1)/\max(n_{\text{points}} - 1, 1)$, where n_{points} and $c2$ are the arguments of the same name. Larger $c2$ gives a smaller radius. Truncation starts at step 2; "none" disables it.

Value

`stein_points()` returns an object of class "stein_points" with:

- X and D: $n_{\text{points}} \times d$ matrices of selected points and their scores.
- ksd: running discrepancy defined above.
- n_eval: evaluation count reported by optimizer at each step; see `fmin_grid()`, `fmin_mc()`, and `fmin_nm()` for what each one counts. The first entry is 1 when x_init is supplied; otherwise it adds 1 for scoring the selected first point. `cum_n_eval = cumsum(n_eval)`.
- method, kernel, truncation, and c2: settings used by the run.

- call: matched function call.

`print(fit)` displays the point-set size, final KSD, and evaluation count, and returns `fit` invisibly. `summary(fit)` returns a `"summary.stein_points"` object with components `method`, `kernel`, `n`, `d`, `evaluation_label`, `n_eval_total`, `call`, `ksd_last`, and `truncation`, plus `c2` when truncation is enabled. `evaluation_label` names what `n_eval_total` counts. Printing that object returns it invisibly.

References

Chen, W. Y., Mackey, L., Gorham, J., Briol, F.-X., and Oates, C. J. (2018). Stein Points. Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 80, 844–853. <https://proceedings.mlr.press/v80/chen18f.html>.

See Also

`fmin_grid()`, `fmin_mc()`, and `fmin_nm()` for the supplied optimizers; `stein_codescent()` to refine a finished point set.

Examples

```
# Target: N(0, 1).
score <- function(X) -X
kernel <- stein_kernel("imq")
opt <- fmin_grid(lb = -4, ub = 4, n0 = 401, grow = FALSE)
fit <- stein_points(score, kernel, n_points = 50, d = 1,
                   optimizer = opt, x_init = 0)

print(fit)
c(mean = mean(fit$X), second_moment = mean(fit$X^2)) # Targets: 0 and 1
```

stein_thinning

Select existing samples by Stein thinning

Description

Selects an ordered sequence of row indices from an existing sample using a greedy Stein-discrepancy criterion. The function does not simulate or move sample points.

Usage

```
stein_thinning(
  X,
  S = NULL,
  m,
  score_function = NULL,
  pre = c("sclmed", "med", "smpcov"),
  kernel = "imq",
  pre_subsample = 1000L,
  pre_subsample_method = c("first", "even", "random")
)
```

Arguments

X	Finite numeric vector or matrix containing n existing samples. Rows are samples and columns are coordinates; a vector is treated as an $n \times 1$ matrix.
S	Optional finite numeric $n \times d$ score matrix with row $s_p(z_i)^\top$. Supply S or score_function.
m	Positive integer number of indices to select.
score_function	Optional function taking X and returning the corresponding $n \times d$ matrix of target scores. When supplied, its result is used instead of S.
pre	Preconditioner: "sclmed", "med", "smpcov", or a symmetric positive-definite $d \times d$ matrix used directly.
kernel	"imq" (default), "gaussian_rbf", or a SteinKernel object. Character inputs use fixed kernel parameters. To set these parameters, supply an object from stein_kernel() . A supplied Gaussian RBF object must have a fixed positive bandwidth.
pre_subsample	For "med" and "sclmed", either a positive integer giving the maximum number of rows used to estimate ρ , Inf for all rows, or an explicit vector of row indices. Ignored for "smpcov" and matrix pre.
pre_subsample_method	Row-selection method used when pre_subsample is scalar: "first" uses the initial rows, "even" uses evenly spaced rows, and "random" samples rows randomly.

Details

Let rows of X be $z_1^\top, \dots, z_n^\top$ and $K_{ab} = k_{0,p}(z_a, z_b)$, with $k_{0,p}$ defined in [stein_kernel_matrix\(\)](#). Scores may be supplied through S or computed once using score_function. If the first $j - 1$ selected indices are $\pi(1), \dots, \pi(j - 1)$, the next index is

$$\pi(j) \in \arg \min_{i \in \{1, \dots, n\}} \left\{ K_{ii} + 2 \sum_{\ell=1}^{j-1} K_{\pi(\ell), i} \right\}.$$

Before the next selection, the chosen Stein-kernel row is added to the running objective. Selection costs $O(nmd)$ after the initial diagonal calculation. Ties use the first minimum. Every row remains eligible, so an index may be selected more than once and m may exceed n ; repeated indices represent repeated mass on the corresponding rows.

pre chooses the matrix M in the kernel distance $r_M(x, y) = (x - y)^\top M(x - y)$. With ρ the median pairwise Euclidean distance among the rows selected by pre_subsample, the three rules are

$$M_{\text{med}} = \frac{1}{\rho^2} I, \quad M_{\text{sclmed}} = \frac{\log(m)}{\rho^2} I, \quad M_{\text{smpcov}} = \widehat{\text{Cov}}(X)^{-1}.$$

"med" and "sclmed" require at least two preconditioning rows, and the default "sclmed" also requires $m > 1$; if $\rho = 0$, ρ^2 is replaced by 1 with a warning. "smpcov" uses all rows and requires a nonsingular empirical covariance matrix.

The default base kernel is

$$k(x, y) = \{1 + r_M(x, y)\}^{-1/2}.$$

A Gaussian RBF kernel must have a fixed positive bandwidth. For supplied built-in RBF or IMQ objects, `pre` replaces a stored preconditioner and warns when the matrices differ. Custom callbacks receive the thinning preconditioner as `M`; see `custom_stein_kernel()` for their interface.

Value

Integer vector $(\pi(1), \dots, \pi(m))$ containing one-based row indices in selection order. It is not sorted and may contain repeated indices. Extract the selected values with `X[idx]` for a vector, or `X[idx, , drop = FALSE]` for a matrix.

References

Riabiz, M., Chen, W. Y., Cockayne, J., Swietach, P., Niederer, S. A., Mackey, L., and Oates, C. J. (2022). Optimal Thinning of MCMC Output. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 84(4), 1059–1081. doi:[10.1111/rssb.12503](https://doi.org/10.1111/rssb.12503).

Examples

```
# Target: N(0, 1).
set.seed(1)
X <- matrix(rnorm(500), ncol = 1)
idx <- stein_thinning(X, score_function = function(X) -X, m = 50)
selected <- X[idx, , drop = FALSE]
c(mean = mean(selected), second_moment = mean(selected^2)) # Targets: 0 and 1
```

svgd

Transport particles with Stein variational gradient descent

Description

Transports an initial set of particles toward a target distribution specified by its score function. SVGD uses a base kernel and its first derivative; it does not construct the pairwise Stein kernel $k_{0,p}$.

Usage

```
svgd(
  x0,
  score_function,
  kernel = stein_kernel(type = "gaussian_rbf"),
  n_iter = 1000,
  step_size = 0.001,
  alpha = 0.9,
  adj_grad = NULL,
  trace_iters = NULL
)
```

Arguments

<code>x0</code>	Finite numeric vector or matrix containing the initial particles. Rows are particles and columns are coordinates; a vector is treated as an $m \times 1$ matrix.
<code>score_function</code>	Function taking the current $m \times d$ particle matrix and returning the corresponding $m \times d$ matrix of target scores.
<code>kernel</code>	A <code>SteinKernel</code> object describing the base kernel. SVGD calls <code>eval_kernel()</code> and <code>grad_x_kernel()</code> , not <code>stein_kernel_matrix()</code> . Use <code>stein_kernel()</code> for a built-in kernel or <code>custom_stein_kernel()</code> to supply the kernel calculations.
<code>n_iter</code>	Nonnegative integer number of particle updates.
<code>step_size</code>	Positive finite update size ϵ . With the default rescaling, each coordinate moves by about <code>step_size</code> per iteration, so <code>n_iter * step_size</code> should be at least the distance the particles need to travel.
<code>alpha</code>	Number in $[0, 1)$ controlling the running squared-gradient average in the default adjustment.
<code>adj_grad</code>	Optional adjustment function. It receives <code>grad</code> , <code>historical_grad</code> , <code>iter</code> , <code>theta</code> , <code>step_size</code> , <code>alpha</code> , and <code>fudge_factor</code> , and must return a finite numeric matrix with the same dimensions as the particles.
<code>trace_iters</code>	Optional integer vector of iterations to retain. Each requested matrix is stored after that iteration's update.

Details

Let $x_1^{(t)}, \dots, x_m^{(t)}$ be the particles at iteration t and $s_p(x) = \nabla_x \log p(x)$. The raw direction for particle i is

$$g_i^{(t)} = \frac{1}{m} \sum_{j=1}^m \left\{ k_\rho(x_j^{(t)}, x_i^{(t)}) s_p(x_j^{(t)}) + \nabla_{x_j} k_\rho(x_j^{(t)}, x_i^{(t)}) \right\}.$$

The first term moves particles toward regions of larger target density; the second repels nearby particles. Each update costs $O(m^2 d)$.

The update is

$$x_i^{(t+1)} = x_i^{(t)} + \epsilon \tilde{g}_i^{(t)},$$

where ϵ is `step_size`. By default, $\tilde{g}$ is the AdaGrad-style rescaling used by the original implementation of Liu and Wang (2016): entrywise,

$$H^{(1)} = g^{(1)} \odot g^{(1)}, \quad H^{(t)} = \alpha H^{(t-1)} + (1 - \alpha) g^{(t)} \odot g^{(t)}, \quad \tilde{g}^{(t)} = \frac{g^{(t)}}{10^{-6} + \sqrt{H^{(t)}}}.$$

Supplying `adj_grad` replaces this rescaling but not the outer multiplication by `step_size`; `adj_grad = function(grad, ...)` gives $\tilde{g}^{(t)} = g^{(t)}$, the plain update as stated by Liu and Wang (2016).

For a Gaussian RBF kernel without a fixed bandwidth, let ρ_t be the median pairwise Euclidean distance between the current particles. The bandwidth is recomputed at every iteration as

$$h_t = \frac{\rho_t}{\sqrt{2 \log m}}.$$

For one particle, $h_t = 1$. If the median is zero or non-finite, `svgd()` stops; supply a fixed positive RBF bandwidth. Fixed-bandwidth RBF and other kernels retain their supplied parameters.

Value

An object of class "svgd" containing:

- X : final $m \times d$ particle matrix.
- D : target scores evaluated at X .
- `trace`: requested post-update matrices named by iteration, or `NULL`.
- `n_eval`, `cum_n_eval`: per-iteration and cumulative score-evaluation counts for the update loop. The final evaluation used for D is not included.
- `kernel`, `n_iter`, and `step_size`: supplied update settings.
- `method` and `call`: method label and matched call.

`print(fit)` displays the particle count, dimension, iteration count, step size, and evaluation count, and returns `fit` invisibly. `summary(fit)` returns a "summary.svgd" object containing the point-set size, kernel, iteration settings, update-loop score-evaluation total, and call. Printing that object returns it invisibly.

References

Liu, Q. and Wang, D. (2016). Stein Variational Gradient Descent: A General Purpose Bayesian Inference Algorithm. *Advances in Neural Information Processing Systems*, 29, 2378–2386. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2016/hash/b3ba8f1bee1238a2f37603d90b58898d-Abstract.html.

Examples

```
# Target: N(0, 1).
score <- function(X) -X
x0 <- matrix(seq(-0.5, 0.5, length.out = 50), ncol = 1)
kernel <- stein_kernel("gaussian_rbf", h = 1)
fit <- svgd(x0, score, kernel = kernel, n_iter = 1000, step_size = 0.01)
ksd <- function(X) {
  sqrt(max(0, sum(stein_kernel_matrix(kernel, X, score(X))))) / nrow(X)
}
c(before = ksd(x0), after = ksd(fit$X))
```

Index

`compute_tau`, 4
`compute_tau()`, 11, 16, 19
`cross_kernel (eval_kernel)`, 6
`cross_kernel()`, 7, 39, 41
`custom_stein_kernel`, 5
`custom_stein_kernel()`, 38, 46, 47

`densitygmm (gmm)`, 21
`densitygmm()`, 21

`eval_kernel`, 6
`eval_kernel()`, 6, 7, 39, 41, 47

`find_median_distance`, 8
`find_median_distance()`, 8, 9, 23, 26, 29
`fmin_grid`, 9
`fmin_grid()`, 9, 10, 36, 43, 44
`fmin_mc (fmin_grid)`, 9
`fmin_mc()`, 9, 10, 36, 43, 44
`fmin_nm (fmin_grid)`, 9
`fmin_nm()`, 9, 10, 36, 43, 44
`fssd_grad_kernel`, 11
`fssd_grad_kernel()`, 6, 14
`fssd_null_pvalue (fssd_statistic)`, 16
`fssd_null_pvalue()`, 4, 5, 14–17, 19
`fssd_opt_test`, 12
`fssd_opt_test()`, 6, 11, 18
`fssd_rand_test`, 14
`fssd_rand_test()`, 18
`fssd_statistic`, 16
`fssd_statistic()`, 4, 5, 17, 19
`fssd_test`, 18
`fssd_test()`, 3, 4, 14–16

`get_score_evaluator`, 20
`get_score_evaluator()`, 21
`gmm`, 21
`gmm()`, 3, 20, 21
`grad_x_kernel (eval_kernel)`, 6
`grad_x_kernel()`, 6, 7, 39, 47

`ksd_u_bootstrap (ksd_u_statistic)`, 24
`ksd_u_bootstrap()`, 23, 25, 26
`ksd_u_statistic`, 24
`ksd_u_statistic()`, 23–26
`ksd_u_test`, 25
`ksd_u_test()`, 3, 8, 23, 25
`ksd_uq_matrix`, 22
`ksd_uq_matrix()`, 24, 26
`ksd_v_bootstrap (ksd_v_statistic)`, 27
`ksd_v_bootstrap()`, 23, 28, 30
`ksd_v_statistic`, 27
`ksd_v_statistic()`, 23, 28, 30
`ksd_v_test`, 29
`ksd_v_test()`, 3, 8, 23, 28
`ksd_vq_matrix (ksd_uq_matrix)`, 22
`ksd_vq_matrix()`, 28, 30

`mala`, 31
`mala()`, 31, 32, 34, 36

`rgmm (gmm)`, 21
`rgmm()`, 21
`rwm (mala)`, 31
`rwm()`, 32, 34, 36

`scale2_kernel (find_median_distance)`, 8
`scale2_kernel()`, 8, 9, 38
`sp_mcmc`, 33
`sp_mcmc()`, 3, 31, 38
`sp_mcmc_eval_candidates (sp_mcmc)`, 33
`sp_mcmc_eval_candidates()`, 32–35
`stats::optim()`, 14
`stein_codescent`, 36
`stein_codescent()`, 9, 10, 43, 44
`stein_kernel`, 38
`stein_kernel()`, 3, 6–8, 40, 41, 45, 47
`stein_kernel_imq_score`, 39
`stein_kernel_imq_score()`, 38
`stein_kernel_inverse_log`, 40
`stein_kernel_inverse_log()`, 38

stein_kernel_matrix, [41](#)
stein_kernel_matrix(), [6](#), [7](#), [26](#), [30](#), [39](#), [45](#),
 [47](#)
stein_points, [42](#)
stein_points(), [3](#), [9](#), [10](#), [34](#), [36–38](#)
stein_thinning, [44](#)
stein_thinning(), [3](#)
steinsampling (steinsampling-package), [3](#)
steinsampling-package, [3](#)
svgd, [46](#)
svgd(), [3](#)

trace_mixed_kernel (eval_kernel), [6](#)
trace_mixed_kernel(), [6](#), [7](#), [39](#), [41](#)