Package: mrddGlobal (via r-universe)

May 28, 2026

Title Global Testing for Multivariate Regression Discontinuity Designs

Version 0.1.0

Description Global testing for regression discontinuity designs with more than one running variable. The function cef_disc_test() is used for testing whether there exist non-zero treatment effects along the boundary of the treated region. The function density_disc_test() is used for testing whether there exist discontinuities in the joint density of the running variables along the boundary of the treated region. The methodology follows Samiahulin (2026), ``Global Testing for Regression Discontinuity Designs with Multiple Running Variables'' <doi:10.48550/arXiv.2602.03819>.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

Imports grf, lpdensity, Rcpp

LinkingTo Rcpp, RcppArmadillo

Suggests dplyr, ggplot2, patchwork, rmarkdown, sf, RANN, knitr

VignetteBuilder knitr

NeedsCompilation yes

Author Artem Samiahulin [aut, cre]

Maintainer Artem Samiahulin <artem.samiahulin@outlook.com>

Config/pak/sysreqs make

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

Date/Publication 2026-03-21 10:20:02 UTC

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

RemoteRef HEAD

RemoteSha 06073f3aac4342185aebd31257b113f73616d735

Contents

`cef_disc_test` *Boundary treatment effect heterogeneity test*

Description

`cef_disc_test` implements the procedure from Samiahulin (2026) that detects the presence of non-zero treatment effects along a treatment boundary in multivariate regression discontinuity design settings.

Usage

```
cef_disc_test(
  X,
  Y,
  t,
  closest_points_all,
  seed = 1,
  K = 2,
  nBoots = 5000,
  num_splits = 1,
  boot_weight_type = "rademacher",
  verbose = TRUE,
  bwselect = "diff",
  ...
)
```

Arguments

`X`	A matrix of running variables, where each row is a separate observation.
`Y`	A numeric vector for the outcome.
`t`	A numeric vector for treatment statuses. Each element should either be zero or one.

`closest_points_all`
: A matrix of the closest boundary points for each observation. This matrix can be created by the user, through the `get_closest_points_bbox` function (with a point cloud), or through the `get_closest_points_shp` function (with a shape file).

`seed`
: A scalar that sets the seed for random number generation.

`K`
: A whole number that represents the number of folds to use for K-fold cross-fitting. Must be greater than or equal to 2. Default is 2.

`nBoots`
: A whole number indicating the number of bootstrap replications to perform when constructing standard errors for the final test statistic. Default is 5000.

`num_splits`
: A whole number indicating the number of splits to try for K-fold cross-fitting. Must be greater than or equal to 1. Default is 1.

`boot_weight_type`
: A character string indicating the type of weights to use in the multiplier bootstrap. The options currently available are "rademacher" (default) and "normal" weights.

`verbose`
: A logical constant indicating whether progress should be printed.

`bwselect`
: Character string indicating whether to use a single common bandwidth ("diff") or two side-specific bandwidths ("each") for the final test statistic. The default is a single common bandwidth. Alternatively, `bwselect` may be a list or numeric vector specifying the bandwidth(s) directly. If a single number is provided, it is used on both sides of the cutoff. If two numbers are provided, the first is used on the left of the cutoff and the second on the right. Bandwidths are constrained to include at least 10 observations on each side of the cutoff and must not extend beyond the maximum support on either side.

`...`
: Further arguments passed to `ll_regression_forest` and `predict.ll_regression_forest`. For more information, see the `grf` package.

Value

Returns a list with the main estimate (`tau`), the bias-corrected estimate (`tau_bc`), standard error for the bias-corrected estimate (`se_tau`), and the p-value for the bias-corrected test statistic (`pval`). Also returns (in `split_df`) right bandwidths by fold/split, left bandwidths by fold/split, main estimates by fold/split, and bias-corrected estimates by fold/split.

References

Samiahulin (2026). *Global Testing in Multivariate Regression Discontinuity Designs*. arXiv Preprint arXiv:2602.03819. https://arxiv.org/abs/2602.03819

Examples

```
# First, let's generate and set up some sample data
set.seed(1)
n <- 1000

x1 <- runif(n, -1, 1)
x2 <- runif(n, -1, 1)
```

```
X <- cbind(x1, x2)
Y <- (x1 + x2) / 3 + rnorm(n, 0, sqrt(0.05))
t <- ifelse(X[,1] >= 0 & X[,2] >= 0, 1, 0)

# Now, let's make a function that computes the closest boundary point.
# For simplicity, let's assume the boundary is defined by min(x1, x2) = 0
closest_boundary_point <- function(df) {
  x1 <- df[, 1]
  x2 <- df[, 2]
  result <- matrix(0, nrow = nrow(df), ncol = 2)

  mask1 <- x1 < 0 & x2 < 0
  result[mask1, ] <- cbind(0, 0)

  mask2 <- x1 >= 0 & x1 > x2 & !mask1
  result[mask2, ] <- cbind(x1[mask2], 0)

  mask3 <- x2 >= 0 & x1 < x2 & !mask1
  result[mask3, ] <- cbind(0, x2[mask3])

  return(result)
}
closest_points <- closest_boundary_point(X)

# With all of the data ready, the cef_disc_test function is ready to be used.
results <- cef_disc_test(X, Y, t, closest_points)
```

`density_disc_test` *Global density test*

Description

`density_disc_test` implements the procedure from Samiahulin (2026) that tests for the presence of discontinuities in the joint density of running variables at the treatment boundary in multivariate regression discontinuity design settings.

Usage

```
density_disc_test(
  X,
  t,
  closest_points_all,
  in_region0,
  in_region1,
  lower_bounds0,
  lower_bounds1,
  upper_bounds0,
```

```
  upper_bounds1,
  mc_samples = 1000,
  seed = 1,
  K = 2,
  nBoots = 5000,
  num_splits = 1,
  bwselect = "diff",
  verbose = FALSE,
  boot_weight_type = "rademacher",
  rand_mid_split = FALSE,
  num_trees = "rot",
  max.depth0 = "rot",
  max.depth1 = "rot"
)
```

Arguments

X	A matrix of running variables, where each row is a separate observation.
t	A numeric vector for treatment statuses. Each element should either be zero or one.
closest_points_all	A matrix of the closest boundary points for each observation. This matrix can be created by the user, through the get_closest_points_bbox function (with a point cloud), or through the get_closest_points_shp function (with a shape file).
in_region0	A function that takes a set of running variables for an observation and returns TRUE when in the control region and FALSE otherwise. FALSE should also be returned when the observation is outside of the support of the running variables.
in_region1	A function that takes a set of running variables for an observation and returns TRUE when in the treated region and FALSE otherwise. FALSE should also be returned when the observation is outside of the support of the running variables.
lower_bounds0	A list of lower bounds for the bounding box around the control region. This list can be created by the user, through the get_closest_points_bbox function (with a point cloud), or through the get_closest_points_shp function (with a shape file).
lower_bounds1	A list of lower bounds for the bounding box around the treated region. This list can be created by the user, through the get_closest_points_bbox function (with a point cloud), or through the get_closest_points_shp function (with a shape file).
upper_bounds0	A list of upper bounds for the bounding box around the control region. This list can be created by the user, through the get_closest_points_bbox function (with a point cloud), or through the get_closest_points_shp function (with a shape file).
upper_bounds1	A list of upper bounds for the bounding box around the treated region. This list can be created by the user, through the get_closest_points_bbox function (with a point cloud), or through the get_closest_points_shp function (with a shape file).

mc_samples	A positive whole number that represents the number of random points to generate to calculate volume in each leaf. Default is 500.
seed	A scalar that sets the seed for random number generation.
K	A whole number that represents the number of folds to use for K-fold cross-fitting. Must be greater than or equal to 2. Default is 2.
nBoots	A whole number indicating the number of bootstrap replications to perform when constructing standard errors for the final test statistic. Must be greater than or equal to 1. Default is 5000.
num_splits	A whole number indicating the number of splits to try for K-fold cross-fitting. Must be greater than or equal to 1. Default is 1.
bwselect	Character string indicating whether to use a single common bandwidth ("diff") or two side-specific bandwidths ("each") for the final test statistic. The default is a single common bandwidth. Alternatively, bwselect may be a list or numeric vector specifying the bandwidth(s) directly. If a single number is provided, it is used on both sides of the cutoff. If two numbers are provided, the first is used on the left of the cutoff and the second on the right. Bandwidths are constrained to include at least 10 observations on each side of the cutoff and must not extend beyond the maximum support on either side.
verbose	A logical constant indicating whether progress should be printed.
boot_weight_type	A character string indicating the type of weights to use in the multiplier bootstrap. The options currently available are "rademacher" (default) and "normal" weights.
rand_mid_split	A logical constant indicating how the trees in the random forest density estimator should be split. If TRUE, make splits using the midpoint between two randomly selected points on an axis. If FALSE (default), make splits using the midpoint of the selected axis. Note that random midpoint splitting is currently an experimental feature that is designed to improve the flexibility and reliability of the original random forest density estimator algorithm. When the boundary is discretized, the deterministic midpoint splits are not as reliable as the random splits.
num_trees	A whole number indicating the number of trees to use for the random forest density estimator. Default is to use a rule-of-thumb based on the sample size and dimension.
max.depth0	A whole number indicating the depth of the trees used to estimate the joint running variable density from the control region. Default is to use a rule-of-thumb based on the sample size and dimension.
max.depth1	A whole number indicating the depth of the trees used to estimate the joint running variable density from the treated region. Default is to use a rule-of-thumb based on the sample size and dimension.

Value

Returns a list with the main estimate (tau), the bias-corrected estimate (tau_bc), standard error for the bias-corrected estimate (se_tau), and the p-value for the bias-corrected test statistic (pval). Also returns (in split_df) right bandwidths by fold/split, left bandwidths by fold/split, main estimates by fold/split, and bias-corrected estimates by fold/split.

References

Samiahulin (2026). *Global Testing in Multivariate Regression Discontinuity Designs*. arXiv Preprint arXiv:2602.03819. https://arxiv.org/abs/2602.03819

Wen, Hongwei, and Hanyuan Hang. 2022. *Random Forest Density Estimation.* In International Conference on Machine Learning, 23701–22. PMLR.

Examples

```
# First, let's generate some data
set.seed(1)
n <- 1000

x1 <- runif(n, -1, 1)
x2 <- runif(n, -1, 1)
X <- cbind(x1, x2)
t <- ifelse(X[,1] >= 0 & X[,2] >= 0, 1, 0)

# Now, let's make a function that computes the closest boundary point.
# For simplicity, let's assume that the boundary is defined by min(x1, x2) = 0
closest_boundary_point <- function(df) {
  x1 <- df[, 1]
  x2 <- df[, 2]
  result <- matrix(0, nrow = nrow(df), ncol = 2)

  mask1 <- x1 < 0 & x2 < 0
  result[mask1, ] <- cbind(0, 0)

  mask2 <- x1 >= 0 & x1 > x2 & !mask1
  result[mask2, ] <- cbind(x1[mask2], 0)

  mask3 <- x2 >= 0 & x1 < x2 & !mask1
  result[mask3, ] <- cbind(0, x2[mask3])

  return(result)
}
closest_points <- closest_boundary_point(X)

# Now let's also define the functions that determine whether an observation is treated or control
in_region1 <- function(x) {
  x[1] >= 0 && x[2] >= 0 && x[1] <= 1 && x[2] <= 1
}

in_region0 <- function(x) {
  !(x[1] >= 0 && x[2] >= 0) && abs(x[1]) <= 1 && abs(x[2]) <= 1
}

# Finally, let's define the bounding boxes for these regions
lower_bounds0 <- c(-1, -1)
lower_bounds1 <- c(0, 0)
upper_bounds0 <- c(1, 1)
upper_bounds1 <- c(1, 1)
```

```
# With all of the data ready, the density_disc_test function is ready to be used
results <- density_disc_test(X, t, closest_points, in_region0, in_region1,
lower_bounds0, lower_bounds1, upper_bounds0, upper_bounds1)
```

`gen_data_cef` *Generates pseudo-random data for experimentation*

Description

Creates pseudo-random outcome data and running variable data for two different data generating processes (DGPs): one that is discontinuous almost everywhere and one that is continuous everywhere.

Usage

```
gen_data_cef(dgp, n = 1000, error_var = 0.05, seed = 1)
```

Arguments

`dgp`	Number (either 1 or 2) that sets the DGP. Use `dgp = 1` for the discontinuous DGP and `dgp = 2` for the continuous DGP. See below for details.
`n`	Number of observations to generate.
`error_var`	Variance of the error term used to create the outcome.
`seed`	Random seed used to reproduce data consistently.

Value

Returns a data frame with the outcome (`Y`) and running variables (`X1` and `X2`) that are independent and follow uniform distributions with supports between -1 and 1. If `dgp = 1`, the outcome is generated as

$$Y = \frac{1}{3}\begin{cases} X_1 + X_2, & \text{if } X_1, X_2 \in [0,1], \\ 1 - X_1, & \text{if } X_1 \in [0,1],\ X_2 \in [-1,0], \\ 1 - X_2, & \text{if } X_1 \in [-1,0],\ X_2 \in [0,1], \\ 1, & \text{if } X_1, X_2 \in [-1,0], \end{cases} + \varepsilon.$$

If `dgp = 2`, the outcome will be generated as

$$Y = \frac{X_1 + X_2}{3} + \varepsilon.$$

In both cases, $\epsilon \sim \mathcal{N}(0, \text{error\_var})$.

Examples

```
# Generate data from the discontinuous DGP
dat <- gen_data_cef(dgp = 1, n = 20000, seed = 1)

# Bin the running variables
nbins <- 50
x1_bins <- cut(dat$X1, nbins)
x2_bins <- cut(dat$X2, nbins)

# Compute binned means of the outcome
Z <- tapply(dat$Y, list(x1_bins, x2_bins), mean)
Z[is.na(Z)] <- 0

# Plot heat map of the conditional expectation
image(
  Z,
  col = terrain.colors(50),
  xlab = expression(X[1]),
  ylab = expression(X[2]),
  main = "Binned Mean of Y (Heat Map)"
)
```

---

gen_data_density *Generates pseudo-random density data for experimentation*

Description

Creates running variable data for two different data generating processes (DGPs): one that is discontinuous almost everywhere and one that is continuous everywhere.

Usage

```
gen_data_density(dgp, n = 1000, seed = 1)
```

Arguments

dgp	Number (either 1 or 2) that sets the DGP. Use dgp = 1 for the discontinuous DGP and dgp = 2 for the continuous DGP. See below for details.
n	Number of observations to generate.
seed	Random seed used to reproduce data consistently.

Value

Returns a matrix with running variables. If dgp = 1, the density of the running variables is generated as

$$f_X(X_1, X_2) = \frac{1}{3} \begin{cases} X_1 + X_2, & \text{if } X_1, X_2 \in [0, 1], \\ 1 - X_1, & \text{if } X_1 \in [0, 1],\ X_2 \in [-1, 0], \\ 1 - X_2, & \text{if } X_1 \in [-1, 0],\ X_2 \in [0, 1], \\ 1, & \text{if } X_1, X_2 \in [-1, 0], \end{cases}$$

If dgp = 2, the outcome will be generated as

$$f_X(X_1, X_2) = \frac{1}{4}$$

Examples

```
# Generate data from the discontinuous density DGP
X1 <- gen_data_density(dgp = 1, n = 20000, seed = 1)

# Bin the running variables
nbins <- 50
x1_bins <- cut(X1[, 1], nbins)
x2_bins <- cut(X1[, 2], nbins)

# Compute empirical density (counts per bin)
Z <- tapply(rep(1, nrow(X1)), list(x1_bins, x2_bins), sum)
Z[is.na(Z)] <- 0
Z <- Z / sum(Z)  # normalize to sum to one

# Plot heat map of the empirical density
image(
  Z,
  col = terrain.colors(50),
  xlab = expression(X[1]),
  ylab = expression(X[2]),
  main = "Empirical Density Heat Map (DGP 1)"
)
```

---

get_closest_points_bbox

Use a point cloud to get closest boundary points and bounding boxes

---

Description

A helper function that takes the point cloud generated by make_cloud and finds the (approximate) closest boundary points and bounding boxes around the treated and control regions.

Usage

```
get_closest_points_bbox(cloud, X, t)
```

Arguments

cloud	A matrix that represents a point cloud of the treatment boundary. This can be generated by the make_cloud function.
X	A matrix of running variables.
t	A numeric vector for treatment indicators. Each element should either be zero or one.

Details

The closest boundary points are found using an Approximate Nearest Neighbors searching algorithm (through the RANN package). The bounding box limits for the treated and control regions are approximated using the smallest box that contains the treated/control observations and all point cloud points.

Value

A matrix of closest boundary points (closest_points), lists of upper and lower bounds for the bounding box of the control region (upper_bounds0 and lower_bounds0 respectively), and lists of upper and lower bounds for the bounding box of the treated region (upper_bounds1 and lower_bounds1 respectively).

Examples

```
# Suppose treatment is assigned when min(x1, x2) >= 0.
# This means that the boundary of the treated region is defined by min(x1, x2) = 0.
set.seed(1)
n <- 1000

# First, let's generate the running variable matrix
x1 <- runif(n, -1, 1)
x2 <- runif(n, -1, 1)
x <- cbind(x1, x2)
t <- ifelse(x[,1] >= 0 & x[,2] >= 0, 1, 0)

# Now, let's create a function that defines the boundary of the treated region
  # This function should equal zero at the boundary
bndry_func <- function(X) {
  # We can add a penalty for having the boundary point outside the set of possible points
  if (any(X < -1 | X > 1)) {
    return(999)
  } else {
    return(min(X))
  }
}
# Now, we can apply the make_cloud function
cloud <- make_cloud(x, bndry_func)

# We can now find the closest boundary points with the get_closest_points_bbox function
df_bndry <- get_closest_points_bbox(cloud, x, t)
closest_points_all <- df_bndry[["closest_points"]]
```

```
# We can extract the limits for the bounding boxes of the treated/control regions
lower_bounds0 <- df_bndry[["lower_bounds0"]]
lower_bounds1 <- df_bndry[["lower_bounds1"]]
upper_bounds0 <- df_bndry[["upper_bounds0"]]
upper_bounds1 <- df_bndry[["upper_bounds1"]]
```

`get_closest_points_shp` *Closest boundary points with a shape file*

Description

Get closest boundary points using a shape file from the `sf` package.

Usage

```
get_closest_points_shp(X, shp_file)
```

Arguments

`X`	A matrix of running variables.
`shp_file`	A shape file that is of class `sf` or `sfc`. Make sure `X` and `shp_file` use the same coordinate system.

Value

A matrix of points that represent closest points to the boundary.

Examples

```
# Suppose treatment is assigned when min(x1, x2) >= 0.
# This means that the boundary of the treated region is defined by min(x1, x2) = 0.
set.seed(1)
n <- 1000

# First, let's generate the running variable matrix
x1 <- runif(n, -1, 1)
x2 <- runif(n, -1, 1)
x <- cbind(x1, x2)

# Loading in shape file data (North Carolina)
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"))
nc <- sf::st_union(nc)

# Generating points in and out of NC
bbox <- sf::st_bbox(nc)
```

```
n <- 2000
pts <- data.frame(
  x = runif(n, bbox["xmin"], bbox["xmax"]),
  y = runif(n, bbox["ymin"], bbox["ymax"])
)
pts_sf <- sf::st_as_sf(pts, coords = c("x", "y"), crs = sf::st_crs(nc))

# Now, we can apply the global_test_prep function
closest_points <- get_closest_points_shp(pts, nc)
```

`make_cloud` *Discretized boundary for data preparation*

Description

Create a point cloud of the boundary based on a boundary level curve.

Usage

```
make_cloud(
  X,
  bndry_func,
  num_pts_discr = 10000,
  start_pt = NULL,
  step_size = 0.1,
  tol = 1e-04,
  seed = 1,
  verbose = TRUE
)
```

Arguments

`X`	A matrix of running variables.
`bndry_func`	A function that represents the level curve of the boundary. This should be an implicit equation that equals zero on the boundary. The input should be a vector and the output should be a real number.
`num_pts_discr`	Number of points used to discretize the boundary.
`start_pt`	Starting point to finding boundary points. If NULL, start at the origin.
`step_size`	Size of the step to use when moving across the running variable space to find boundary points.
`tol`	Distance from the true boundary where a point is considered a boundary point.
`seed`	Random seed used to reproduce data consistently.
`verbose`	If `TRUE`, print the progress of the point cloud generating algorithm.

Details

The algorithm used to find the closest boundary points begins by selecting a starting point. From the starting point, a new trial point is generated by perturbing the coordinates with normal noise of mean zero and standard deviation `step_size`.

If the new trial point is within a small distance of the boundary, it is saved as a boundary point and used as the new current point.

If the new trial point is not within the tolerance distance of the boundary but is closer to it, the trial point becomes the new current point.

If the new trial point is further from the boundary, it is discarded.

This process repeats until `num_pts_discr` boundary points are found.

Value

A matrix of points that represent a point cloud of the boundary.

Examples

```
# Suppose treatment is assigned when min(x1, x2) >= 0.
# This means that the boundary of the treated region is defined by min(x1, x2) = 0.
set.seed(1)
n <- 1000

# First, let's generate the running variable matrix
x1 <- runif(n, -1, 1)
x2 <- runif(n, -1, 1)
x <- cbind(x1, x2)

# Now, let's create a function that defines the boundary of the treated region
  # This function should equal zero at the boundary
bndry_func <- function(X) {
  # We can add a penalty for having the boundary point outside the set of possible points
  if (any(X < -1 | X > 1)) {
    return(999)
  } else {
    return(min(X))
  }
}

# Finally, we can apply the make_cloud function
cloud <- make_cloud(x, bndry_func)
```

`print.cef_disc_test` *Print global treatment effect heterogeneity test results*

Description

Print global treatment effect heterogeneity test results

Usage

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

Arguments

x	An object of class "cef_disc_test"
...	Further arguments (unused)

Value

Invisibly returns the input object x. This function is called for its side effect, which is to print a formatted summary of the global treatment effect heterogeneity test results to the console, including the main estimate, bias-corrected estimate, standard error, and p-value.

---

print.density_disc_test

Print global density test results

Description

Print global density test results

Usage

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

Arguments

x	An object of class "density_disc_test"
...	Further arguments (unused)

Value

Invisibly returns the input object x. This function is called for its side effect, which is to print a formatted summary of the global density test results to the console, including the main estimate, bias-corrected estimate, standard error, and p-value.

summary.cef_disc_test *Summarize global treatment effect heterogeneity test results*

Description

Summarize global treatment effect heterogeneity test results

Usage

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

Arguments

object	An object of class "cef_disc_test"
...	Further arguments (unused)

Value

Invisibly returns the input object object. This function is called for its side effect of printing a detailed summary of the heterogeneity test results to the console, including both global estimates and a truncated view of split- and fold-level results.

summary.density_disc_test
Summarize global density test results

Description

Summarize global density test results

Usage

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

Arguments

object	An object of class "density_disc_test"
...	Further arguments (unused)

Value

Invisibly returns the input object object. This function is called for its side effect of printing a detailed summary of the density test results to the console, including both global estimates and a truncated view of split- and fold-level results.

Index