

Package ‘causalOT’

May 8, 2026

Type Package

Title Optimal Transport Weights for Causal Inference

Version 1.0.4

Date 2026-03-07

Author Eric Dunipace [aut, cre] (ORCID:
<<https://orcid.org/0000-0001-8909-213X>>)

Maintainer Eric Dunipace <edunipace@mail.harvard.edu>

Description Uses optimal transport distances to find probabilistic matching estimators for causal inference. These methods are described in Dunipace, Eric (2021) <[doi:10.48550/arXiv.2109.01991](https://doi.org/10.48550/arXiv.2109.01991)>. The package will build the weights, estimate treatment effects, and calculate confidence intervals via the methods described in the paper. The package also supports several other methods as described in the help files.

License GPL (== 3.0)

Imports CBPS, ggplot2, lbfgsb3c, loo, Matrix (>= 1.5-0), matrixStats, methods, osqp, R6 (>= 2.4.1), Rcpp (>= 1.0.3), rlang, sandwich, torch, utils

LinkingTo BH (>= 1.66.0), Rcpp (>= 0.12.0), RcppEigen (>= 0.3.3.3.0), torch

Suggests data.table (>= 1.12.8), testthat (>= 2.1.0), knitr, reticulate, rkeops (>= 2.2.2), rmarkdown, V8, withr

Additional_repositories <https://ericdunipace.github.io/drat/>

Biarch true

Depends R (>= 4.1.0)

Encoding UTF-8

RoxygenNote 7.3.3

LazyData true

VignetteBuilder knitr, rmarkdown

Collate 'DataSimClass.R' 'dataHolder.R' 'weightsClass.R' 'ESS.R'
 'OT.R' 'PSIS.R' 'RcppExports.R' 'balanceFunctions.R'
 'barycentricProjection.R' 'calc_weight.R' 'causalOT-package.R'
 'cost_functions.R' 'scmClass.R' 'gridSearch.R' 'cotClass.R'
 'cotOOP.R' 'cot_opts.R' 'likelihoodClass.R' 'mean_balance.R'
 'summary.R' 'supportedMethods.R' 'treatment_effect.R' 'utils.R'
 'zzz.R'

NeedsCompilation yes

Repository CRAN

Date/Publication 2026-03-10 07:00:02 UTC

Contents

| | |
|------------------------|----|
| barycentric_projection | 3 |
| calc_weight | 5 |
| causalWeights-class | 7 |
| coef.causalEffect | 8 |
| cotOptions | 9 |
| CRASH3 | 12 |
| dataHolder | 14 |
| DataSim | 15 |
| df2dataHolder | 17 |
| entBWOptions | 18 |
| ESS | 19 |
| estimate_effect | 20 |
| Hainmueller | 21 |
| LaLonde | 23 |
| mean_balance | 26 |
| Measure | 26 |
| OTProblem | 30 |
| ot_distance | 43 |
| plot.causalWeights | 46 |
| pph | 47 |
| predict.bp | 48 |
| print.dataHolder | 50 |
| PSIS | 50 |
| sbwOptions | 52 |
| scmOptions | 53 |
| summary.causalWeights | 53 |
| supported_methods | 55 |
| vcov.causalEffect | 55 |

Index

57

 barycentric_projection

Barycentric Projection outcome estimation

Description

Barycentric Projection outcome estimation

Usage

```
barycentric_projection(
  formula,
  data,
  weights,
  separate.samples.on = "z",
  penalty = NULL,
  cost_function = NULL,
  p = 2,
  debias = FALSE,
  cost.online = "auto",
  diameter = NULL,
  niter = 1000L,
  tol = 1e-07,
  ...
)
```

Arguments

| | |
|---------------------|---|
| formula | A formula object specifying the outcome and covariates. |
| data | A data.frame of the data to use in the model. |
| weights | Either a vector of weights, one for each observations, or an object of class causalWeights . |
| separate.samples.on | The variable in the data denoting the treatment indicator. How to separate samples for the optimal transport calculation |
| penalty | The penalty parameter to use in the optimal transport calculation. By default it is $1/\log(n)$. |
| cost_function | A user supplied cost function. If supplied, must take arguments x1, x2, and p. |
| p | The power to raise the cost function. Default is 2.0. For user supplied cost functions, the cost will not be raised by this power unless the user so specifies. |
| debias | Should debiased barycentric projections be used? See details. |
| cost.online | Should an online cost algorithm be used? Default is "auto", which selects an online cost algorithm when the sample size in each group specified by separate.samples.on, n_0 and n_1 , is such that $n_0 \cdot n_1 \geq 5000^2$ Must be one of "auto", "online", or "tensorized". The last of these is the offline option. |

| | |
|----------|--|
| diameter | The diameter of the covariate space, if known. |
| niter | The maximum number of iterations to run the optimal transport problems |
| tol | The tolerance for convergence of the optimal transport problems |
| ... | Not used at this time. |

Details

The barycentric projection uses the dual potentials from the optimal transport distance between the two samples to calculate projections from one sample into another. For example, in the sample of controls, we may wish to know their outcome had they been treated. In general, we then seek to minimize

$$\operatorname{argmin}_{\eta} \sum_{ij} \operatorname{cost}(\eta_i, y_j) \pi_{ij}$$

where π_{ij} is the primal solution from the optimal transport problem.

These values can also be de-biased using the solutions from running an optimal transport problem of one sample against itself. Details are listed in Pooladian et al. (2022) <https://arxiv.org/abs/2202.08919>.

Value

An object of class "bp" which is a list with slots:

- potentials The dual potentials from calculating the optimal transport distance
- penalty The value of the penalty parameter used in calculating the optimal transport distance
- cost_function The cost function used to calculate the distances between units.
- cost_alg A character vector denoting if an L_1 distance, a squared euclidean distance, or other distance metric was used.
- p The power to which the cost matrix was raised if not using a user supplied cost function.
- debias Whether barycentric projections should be debiased.
- tensorized TRUE/FALSE denoting whether to use offline cost matrices.
- data An object of class `dataHolder` with the data used to calculate the optimal transport distance.
- y_a The outcome vector in the first sample.
- y_b The outcome vector in the second sample.
- x_a The covariate matrix in the first sample.
- x_b The covariate matrix in the second sample.
- a The empirical measure in the first sample.
- b The empirical measure in the second sample.
- terms The terms object from the formula.

Examples

```
if(torch::torch_is_installed()) {
  set.seed(23483)
  n <- 2^5
  pp <- 6
  overlap <- "low"
  design <- "A"
  estimate <- "ATT"
  power <- 2
  data <- causalOT::Hainmueller$new(n = n, p = pp,
  design = design, overlap = overlap)

  data$gen_data()

  weights <- causalOT::calc_weight(x = data,
  z = NULL, y = NULL,
  estimand = estimate,
  method = "NNM")

  df <- data.frame(y = data$get_y(), z = data$get_z(), data$get_x())

  fit <- causalOT::barycentric_projection(y ~ ., data = df,
  weight = weights,
  separate.samples.on = "z",
  niter = 2)
  inherits(fit, "bp")
}
```

calc_weight

Estimate causal weights

Description

Estimate causal weights

Usage

```
calc_weight(
  x,
  z,
  estimand = c("ATC", "ATT", "ATE"),
  method = supported_methods(),
  options = NULL,
  weights = NULL,
  ...
)
```

Arguments

| | |
|----------|--|
| x | A numeric matrix of covariates. You can also pass an object of class dataHolder or DataSim , which will make argument z not necessary, |
| z | A binary treatment indicator. |
| estimand | The estimand of interest. One of "ATT", "ATC", or "ATE". |
| method | The method to estimate the causal weights. Must be one of the methods returned by supported_methods() . |
| options | The options for the solver. Specific options depend on the solver you will be using and you can use the solver specific options functions as detailed below.. |
| weights | The sample weights. Should be NULL or have a weight for each observation in the data. Normalized to sum to one. |
| ... | Not used at this time. |

Details

We detail some of the particulars of the function arguments below.

Causal Optimal Transport (COT):

This is the main method of the package. This method relies on various solvers depending on the particular options chosen. Please see [cotOptions\(\)](#) for more details.

Energy Balancing Weights (EnergyBW):

This is equivalent to COT with an infinite penalty parameter, `options(lambda = Inf)`. Uses the same solver and options as COT, [cotOptions\(\)](#).

Nearest Neighbor Matching with replacement (NNM):

This is equivalent to COT with a penalty parameter = 0, `options(lambda = 0)`. Uses the same solver and options as COT, [cotOptions\(\)](#).

Synthetic Control Method (SCM):

The SCM method is equivalent to an OT problem from a different angle. See [scmOptions\(\)](#).

Entropy Balancing Weights (EntropyBW):

This method balances chosen functions of the covariates specified in the data argument, x. See [entBWOptions\(\)](#) for more details. Hainmueller (2012).

Stable Balancing Weights (SBW):

Entropy Balancing Weights with a different penalty parameter, proposed by Zuizarreta (2012). See [sbwOptions\(\)](#) for more details

Covariate Balancing Propensity Score (CBPS):

The CBPS method of Imai and Ratkovic. Options argument is passed to the function [CBPS\(\)](#).

Logistic Regression or Probit Regression:

The main methods historically for implementing inverse probability weights. Options are passed directly to the `glm` function from R.

Value

An object of class `causalWeights`

See Also

`estimate_effect()`

Examples

```
set.seed(23483)
n <- 2^5
p <- 6
#### get data ####
data <- Hainmueller$new(n = n, p = p)
data$gen_data()
x <- data$get_x()
z <- data$get_z()

if (torch::torch_is_installed()) {
  # estimate weights
  weights <- calc_weight(x = x,
                        z = z,
                        estimand = "ATE",
                        method = "COT",
                        options = list(lambda = 0))
  #we can also use the dataSim object directly
  weightsDS <- calc_weight(x = data,
                          z = NULL,
                          estimand = "ATE",
                          method = "COT",
                          options = list(lambda = 0))
  all.equal(weights@w0, weightsDS@w0)
  all.equal(weights@w1, weightsDS@w1)
}
```

causalWeights-class *causalWeights class*

Description

causalWeights class

Details

This object is returned by the `calc_weight` function in this package. The slots can be accessed as any S4 object. There is no publicly accessible constructor function.

Slots

- w0 A slot with the weights for the control group with n_0 entries. Weights sum to 1.
- w1 The weights for the treated group with n_1 entries. Weights sum to 1.
- estimand A character denoting the estimand targeted by the weights. One of "ATT", "ATC", or "ATE".
- info A slot to store a variety of info for inference. Currently under development.
- method A character denoting the method used to estimate the weights.
- penalty A list or the selected penalty parameters, if relevant.
- data The dataHolder object containing the original data.
- call The call used to construct the weights.

| | |
|--------------------------------|--|
| <code>coef.causalEffect</code> | <i>Extract treatment effect estimate</i> |
|--------------------------------|--|

Description

Extract treatment effect estimate

Usage

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

Arguments

| | |
|---------------------|--|
| <code>object</code> | An object of class <code>causalEffect</code> |
| <code>...</code> | Not used |

Value

A number corresponding to the estimated treatment effect

Examples

```
# set-up data
set.seed(1234)
data <- Hainmueller$new()
data$gen_data()

# calculate quantities
weight <- calc_weight(data, method = "Logistic", estimand = "ATE")
tx_eff <- estimate_effect(causalWeights = weight)

all.equal(coef(tx_eff), c(estimate = tx_eff@estimate))
```

cotOptions

*Options available for the COT method***Description**

Options available for the COT method

Usage

```

cotOptions(
  lambda = NULL,
  delta = NULL,
  opt.direction = c("dual", "primal"),
  debias = TRUE,
  p = 2,
  cost.function = NULL,
  cost.online = "auto",
  diameter = NULL,
  balance.formula = NULL,
  quick.balance.function = TRUE,
  grid.length = 7L,
  torch.optimizer = torch::optim_rmsprop,
  torch.scheduler = torch::lr_multiplicative,
  niter = 2000,
  nboot = 100L,
  lambda.bootstrap = 0.05,
  tol = 1e-04,
  device = NULL,
  dtype = NULL,
  ...
)

```

Arguments

| | |
|---------------|---|
| lambda | The penalty parameter for the entropy penalized optimal transport. Default is NULL. Can be a single number or a set of numbers to try. |
| delta | The bound for balancing functions if they are being used. Only available for biased entropy penalized optimal transport. Can be a single number or a set of numbers to try. |
| opt.direction | Should the optimizer solve the primal or dual problems. Should be one of "dual" or "primal" with a default of "dual" since it is typically faster. |
| debias | Should debiased optimal transport be used? TRUE or FALSE. |
| p | The power of the cost function to use for the cost. |
| cost.function | A function to calculate the pairwise costs. Should take arguments x1, x2, and p. Default is NULL. |

| | |
|-------------------------------------|--|
| <code>cost.online</code> | Should an online cost algorithm be used? One of "auto", "online", or "tensorized". "tensorized" is the offline option. |
| <code>diameter</code> | The diameter of the covariate space, if known. Default is NULL. |
| <code>balance.formula</code> | Formula for the balancing functions. |
| <code>quick.balance.function</code> | TRUE or FALSE denoting whether balance function constraints should be selected via a linear program (TRUE) or just checked for feasibility (FALSE). Default is TRUE. |
| <code>grid.length</code> | The number of penalty parameters to explore in a grid search if none are provided in arguments <code>lambda</code> or <code>delta</code> . |
| <code>torch.optimizer</code> | The torch optimizer to use for methods using debiased entropy penalized optimal transport. If <code>debiased</code> is FALSE or <code>opt.direction</code> is "primal", will default to <code>torch::optim_lbfgs()</code> . Otherwise <code>torch::optim_rmsprop()</code> is used. |
| <code>torch.scheduler</code> | The scheduler for the optimizer. Defaults to <code>torch::lr_multiplicative()</code> . |
| <code>niter</code> | The number of iterations to run the solver |
| <code>nboot</code> | The number of iterations for the bootstrap to select the final penalty parameters. |
| <code>lambda.bootstrap</code> | The penalty parameter to use for the bootstrap hyperparameter selection of <code>lambda</code> . |
| <code>tol</code> | The tolerance for convergence |
| <code>device</code> | An object of class <code>torch_device</code> denoting which device the data will be located on. Default is NULL which will try to use a gpu if available. |
| <code>dtype</code> | An object of class <code>torch_dtype</code> that determines data type of the data, i.e. double, float, integer. Default is NULL which will try to select for you. |
| <code>...</code> | Arguments passed to the solvers. See details |

Value

A list of class `cotOptions` with the following slots

- `lambda` The penalty parameter for the optimal transport distance
- `delta` The constraint for the balancing functions
- `opt.direction` Whether to solve the primal or dual optimization problems
- `debias` TRUE or FALSE if debiased optimal transport distances are used
- `balance.formula` The formula giving how to generate the balancing functions.
- `quick.balance.function` TRUE or FALSE whether quick balance functions will be run.
- `grid.length` The number of parameters to check in a grid search of best parameters
- `p` The power of the cost function
- `cost.online` Whether online costs are used
- `cost.function` The user supplied cost function if supplied.

- `diameter` The diameter of the covariate space.
- `torch.optimizer` The torch optimizer used for Sinkhorn Divergences
- `torch.scheduler` The scheduler for the torch optimizer
- `solver.options` The arguments to be passed to the torch optimizer
- `scheduler.options` The arguments to be passed to the torch scheduler
- `osqp.options` Arguments passed to the osqp function if quick balance functions are used.
- `niter` The number of iterations to run the solver
- `nboot` The number of bootstrap samples
- `lambda.bootstrap` The penalty parameter to use for the bootstrap hyperparameter selection.
- `tol` The tolerance for convergence.
- `device` An object of class torch_device.
- `dtype` An object of class torch_dtype.

Solvers and distances

The function is setup to direct the COT optimizer to run two basic methods: debiased entropy penalized optimal transport (Sinkhorn Divergences) or entropy penalized optimal transport (Sinkhorn Distances).

Sinkhorn Distances:

The optimal transport problem solved is $\min_w OT_\lambda(w, b)$ where

$$OT_\lambda(w, b) = \sum_{ij} C(x_i, x_j) P_{ij} + \lambda \sum_{ij} P_{ij} \log(P_{ij}),$$

such that the rows of the matrix P_{ij} sum to w and the columns sum to b . In this case $C(\cdot, \cdot)$ is the cost between units i and j .

Sinkhorn Divergences:

The Sinkhorn Divergence solves

$$\min_w OT_\lambda(w, b) - 0.5 OT_\lambda(w, w) - 0.5 * OT_\lambda(b, b).$$

The solver for this function uses the torch package in R and by default will use the `optim_rmsprop` solver. Your desired torch optimizer can be passed via `torch.optimizer` with a scheduler passed via `torch.scheduler`. GPU support is available as detailed in the torch package. Additional arguments in `...` are passed as extra arguments to the torch optimizer and schedulers as appropriate.

Function balancing

There may be certain functions of the covariates that we wish to balance within some tolerance, δ . For these functions B , we will desire

$$\frac{\sum_{i:Z_i=0} w_i B(x_i) - \sum_{j:Z_j=1} B(x_j)/n_1}{\sigma} \leq \delta$$

, where in this case we are targeting balance with the treatment group for the ATT. σ is the pooled standard deviation prior to balancing.

Cost functions

The cost function specifies pairwise distances. If argument `cost.function` is `NULL`, the function will default to using L_p^p distances with a default $p = 2$ supplied by the argument `p`. So for $p = 2$, the cost between units x_i and x_j will be

$$C(x_i, x_j) = \frac{1}{2} \|x_i - x_j\|_2^2.$$

If `cost.function` is provided, it should be a function that takes arguments `x1`, `x2`, and `p`: `function(x1, x2, p){...}`.

Examples

```
if ( torch::torch_is_installed() ) {
  opts1 <- cotOptions(lambda = 1e3, torch.optimizer = torch::optim_rmsprop)
  opts2 <- cotOptions(lambda = NULL)
  opts3 <- cotOptions(lambda = seq(0.1, 100, length.out = 7))
}
```

CRASH3

CRASH3 data example

Description

CRASH3 data example

CRASH3 data example

Details

Returns the CRASH3 data. Note that `gen_data()` will initialize the fixed data for `x` and `y`, but `z` is generated from `Binom(0.5)`.

Value

An [R6](#) object of class [DataSim](#)

Super class

`causalOT::DataSim` -> CRASH3

Public fields

`site_id` The site of the observation in terms of the original RCT.

Methods**Public methods:**

- `CRASH3$gen_data()`
- `CRASH3$gen_x()`
- `CRASH3$gen_y()`
- `CRASH3$gen_z()`
- `CRASH3$new()`
- `CRASH3$clone()`

Method `gen_data()`: The site ID for the observations
Draws new treatment indicators. x and y data are fixed.

Usage:

```
CRASH3$gen_data()
```

Method `gen_x()`: Sets up the covariate data. This data is fixed.

Usage:

```
CRASH3$gen_x()
```

Method `gen_y()`: Sets up the outcome data. This data is fixed.

Usage:

```
CRASH3$gen_y()
```

Method `gen_z()`: Sets up the treatment indicator. Drawn as $Z \sim \text{Binom}(0.5)$

Usage:

```
CRASH3$gen_z()
```

Method `new()`: Initializes the CRASH3 object.

Usage:

```
CRASH3$new(n = NULL, p = NULL, param = list(), design = NA_character_, ...)
```

Arguments:

n Not used. Maintained for symmetry with other DataSim objects.

p Not used. Maintained for symmetry with other DataSim objects.

param Not used. Maintained for symmetry with other DataSim objects.

design Not used

... Not used.

Examples:

```
crash <- CRASH3$new()
crash$gen_data()
crash$get_n()
crash$site_id
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
CRASH3$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
## -----
## Method `CRASH3$new`
## -----

crash <- CRASH3$new()
crash$gen_data()
crash$get_n()
crash$site_id
```

dataHolder

dataHolder

Description

dataHolder

Usage

```
dataHolder(x, z, y = NA_real_, weights = NA_real_)
```

Arguments

| | |
|---------|--|
| x | the covariate data. Can be a matrix, an object of class dataHolder or a DataSim object. The latter two object types won't need arguments z or y. |
| z | the treatment indicator |
| y | the outcome data |
| weights | the empirical distribution of the sample |

Details

Creates an object used internally by the causalOT package for data management.

Value

Returns an object of class dataHolder with slots

- x matrix. A matrix of confounders.
- z integer. The treatment indicator, $z_i \in \{0, 1\}$.
- y numeric. The outcome data.
- n0 integer. The number of observations where $z==0$
- n1 integer. The number of observations where $z==1$
- weights numeric. The empirical distribution of the full sample.

Examples

```
x <- matrix(0, 100, 10)
z <- stats::rbinom(100, 1, 0.5)

# don't need to provide outcome
# function will assume each observation gets equal mass
dataHolder(x = x, z = z)
```

DataSim

R6 Data Generating Parent Class

Description

R6 Data Generating Parent Class

R6 Data Generating Parent Class

Details

Can be used to make your own data simulation class. Should have the same slots listed in this class at a minimum, but you can add your own, of course. An easy way to do this is to make your class inherit from this one. See the example.

Value

An [R6](#) object

Methods

Public methods:

- [DataSim\\$get_x\(\)](#)
- [DataSim\\$get_y\(\)](#)
- [DataSim\\$get_z\(\)](#)
- [DataSim\\$get_n\(\)](#)
- [DataSim\\$get_x1\(\)](#)
- [DataSim\\$get_x0\(\)](#)
- [DataSim\\$get_p\(\)](#)
- [DataSim\\$get_tau\(\)](#)
- [DataSim\\$gen_data\(\)](#)
- [DataSim\\$clone\(\)](#)

Method [get_x\(\)](#): Gets the covariate data

Usage:

`DataSim$get_x()`

Method [get_y\(\)](#): Gets the outcome vector

Usage:

DataSim\$get_y()

Method get_z(): Gets the treatment indicator

Usage:

DataSim\$get_z()

Method get_n(): Gets the number of observations

Usage:

DataSim\$get_n()

Method get_x1(): Gets the covariate data for the treated individuals

Usage:

DataSim\$get_x1()

Method get_x0(): Gets the covariate data for the control individuals

Usage:

DataSim\$get_x0()

Method get_p(): Gets the dimensionality covariate data

Usage:

DataSim\$get_p()

Method get_tau(): Gets the individual treatment effects

Usage:

DataSim\$get_tau()

Method gen_data(): Generates the data. Default is an empty function

Usage:

DataSim\$gen_data()

Method clone(): The objects of this class are cloneable with this method.

Usage:

DataSim\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
MyClass <- R6::R6Class("MyClass",  
  inherit = DataSim,  
  public = list(),  
  private = list())
```

| | |
|---------------|----------------------|
| df2dataHolder | <i>df2dataHolder</i> |
|---------------|----------------------|

Description

Function to turn a `data.frame` into a `dataHolder` object.

Usage

```
df2dataHolder(  
  treatment.formula,  
  outcome.formula = NA_character_,  
  data,  
  weights = NA_real_  
)
```

Arguments

| | |
|--------------------------------|--|
| <code>treatment.formula</code> | a formula specifying the treatment indicator and covariates. Required. |
| <code>outcome.formula</code> | an optional formula specifying the outcome function. |
| <code>data</code> | a <code>data.frame</code> with the data |
| <code>weights</code> | optional vector of sampling weights for the data |

Details

This will take the formulas specified and transform that `data.frame` into a `dataHolder` object that is used internally by the `causalOT` package. Take care if you do not specify an outcome formula that you do not include the outcome in the `data.frame`. If you are not careful, the function may include the outcome as a covariate, which is not kosher in causal inference during the design phase.

If both `outcome.formula` and `treatment.formula` are specified, it will assume you are in the design phase, and create a combined covariate matrix to balance on the assumed treatment and outcome models.

If you are in the outcome phase of estimation, you can just provide a dummy formula for the `treatment.formula` like "z ~ 0" just so the function can identify the treatment indicator appropriately in the data creation phase.

Value

Returns an object of class `dataHolder()`

Examples

```

set.seed(20348)
n <- 15
d <- 3
x <- matrix(stats::rnorm(n*d), n, d)
z <- rbinom(n, 1, prob = 0.5)
y <- rnorm(n)
weights <- rep(1/n,n)
df <- data.frame(x, z, y)
dh <- df2dataHolder(
  treatment.formula = "z ~ .",
  outcome.formula = "y ~ ." ,
  data = df,
  weights = weights)

```

entBWOptions

Options for the Entropy Balancing Weights

Description

Options for the Entropy Balancing Weights

Usage

```
entBWOptions(delta = NULL, grid.length = 20L, nboot = 1000L, ...)
```

Arguments

| | |
|-------------|--|
| delta | A number or vector of tolerances for the balancing functions. Default is NULL which will use a grid search |
| grid.length | The number of values to try in the grid search |
| nboot | The number of bootstrap samples to run during the grid search. |
| ... | Arguments passed on to lbfgsb3c() |

Value

A list of class entBWOptions with slots

- delta Delta values to try
- grid.length The number of parameters to try
- nboot Number of bootstrap samples
- solver.options A list of options passed to ['lbfgsb3c\(\)](#)

Function balancing

This method will balance functions of the covariates within some tolerance, δ . For these functions B , we will desire

$$\frac{\sum_{i:Z_i=0} w_i B(x_i) - \sum_{j:Z_j=1} B(x_j)/n_1}{\sigma} \leq \delta$$

, where in this case we are targeting balance with the treatment group for the ATT. σ is the pooled standard deviation prior to balancing.

Examples

```
opts <- entBWOptions(delta = 0.1)
```

| | |
|-----|------------------------------|
| ESS | <i>Effective Sample Size</i> |
|-----|------------------------------|

Description

Effective Sample Size

Usage

```
ESS(x)

## S4 method for signature 'numeric'
ESS(x)

## S4 method for signature 'causalWeights'
ESS(x)
```

Arguments

`x` Either a vector of weights summing to 1 or an object of class `causalWeights`

Details

Calculates the effective sample size as described by Kish (1965). However, this calculation has some problems and the `PSIS()` function should be used instead.

Value

Either a number denoting the effective sample size or if `x` is of class `causalWeights`, then returns a list of both values in the treatment and control groups.

Methods (by class)

- `ESS(numeric)`: default ESS method for numeric vectors
- `ESS(causalWeights)`: ESS method for objects of class `causalWeights`

See Also[PSIS\(\)](#)**Examples**

```
x <- rep(1/100,100)
ESS(x)
```

| | |
|-----------------|-----------------------------------|
| estimate_effect | <i>Estimate treatment effects</i> |
|-----------------|-----------------------------------|

Description

Estimate treatment effects

Usage

```
estimate_effect(
  causalWeights,
  x = NULL,
  y = NULL,
  model.function,
  estimate.separately = TRUE,
  augment.estimate = FALSE,
  normalize.weights = TRUE,
  ...
)
```

Arguments

| | |
|---------------------|---|
| causalWeights | An object of class causalWeights |
| x | A dataHolder, matrix, data.frame, or object of class DataSim. See calc_weight for more details how to input the data. If NULL, will use the info in the causalWeights argument. |
| y | The outcome vector. |
| model.function | The modeling function to use, if desired. Must take arguments "formula", "data", and "weights". Other arguments passed via ..., the dots. |
| estimate.separately | Should the outcome model be estimated separately in each treatment group? TRUE or FALSE. |
| augment.estimate | Should an augmented, doubly robust estimator be used? |
| normalize.weights | Should the weights in the causalWeights argument be normalized to sum to one prior to effect estimation? |
| ... | Pass additional arguments to the outcome modeling functions. |

Value

an object of class `causalEffect`

Examples

```
if ( torch::torch_is_installed() ){
# set-up data
data <- Hainmueller$new()
data$gen_data()

# calculate quantities
weight <- calc_weight(data, method = "COT",
                      estimand = "ATT",
                      options = list(lambda = 0))
tx_eff <- estimate_effect(causalWeights = weight)

# get estimate
print(tx_eff@estimate)
all.equal(coef(tx_eff), c(estimate = tx_eff@estimate))
}
```

Hainmueller

Hainmueller data example

Description

Hainmueller data example

Hainmueller data example

Details

Generates the data as described in Hainmueller (2012).

Value

An R6 object of class `DataSim`

Super class

`causalOT::DataSim` -> Hainmueller

Methods**Public methods:**

- `Hainmueller$gen_data()`
- `Hainmueller$gen_x()`
- `Hainmueller$gen_y()`

- `Hainmueller$gen_z()`
- `Hainmueller$new()`
- `Hainmueller$get_design()`
- `Hainmueller$get_pscore()`
- `Hainmueller$clone()`

Method `gen_data()`: Generates the data

Usage:

```
Hainmueller$gen_data()
```

Method `gen_x()`: Generates the covariate data

Usage:

```
Hainmueller$gen_x()
```

Method `gen_y()`: Generates the outcome data

Usage:

```
Hainmueller$gen_y()
```

Method `gen_z()`: Generates the treatment indicator

Usage:

```
Hainmueller$gen_z()
```

Method `new()`: Generates the the Hainmueller R6 class

Usage:

```
Hainmueller$new(
  n = 100,
  p = 6,
  param = list(),
  design = "A",
  overlap = "low",
  ...
)
```

Arguments:

`n` The number of observations

`p` The dimensions of the covariates. Fixed to 6.

`param` The data generating parameters fed as a list.

`design` One of "A" or "B". See details.

`overlap` One of "high", "low", or "medium". See details.

`...` Extra arguments. Currently unused.

Details:

Design:

Design "A" is the setting where the outcome is generated from a linear model, $Y(0) = Y(1) = X_1 + X_2 + X_3 - X_4 + X_5 + X_6 + \eta$ and design "B" is where the outcome is generated from the non-linear model $Y(0) = Y(1) = (X_1 + X_2 + X_5)^2 + \eta$.

Overlap:

The treatment indicator is generated from $Z = 1(X_1 + 2X_2 - 2X_3 - X_4 - 0.5X_5 + X_6 + \nu > 0)$, where ν depends on the overlap selected. If overlap is "high", then $\nu \sim N(0, 100)$. If overlap is "low", then $\nu \sim N(0, 30)$. Finally, if overlap is "medium", then ν is drawn from a χ^2 with 5 degrees of freedom that is scaled and centered to have mean 0.5 and variance 67.6.

Returns: An object of class `DataSim`.

Examples:

```
data <- Hainmueller$new(n = 100, p = 6, design = "A", overlap = "low")
data$gen_data()
print(data$get_x()[1:2,])
```

Method `get_design()`: Returns the chosen design parameters

Usage:

```
Hainmueller$get_design()
```

Method `get_pscore()`: Returns the true propensity score

Usage:

```
Hainmueller$get_pscore()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
Hainmueller$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## -----
## Method `Hainmueller$new`
## -----

data <- Hainmueller$new(n = 100, p = 6, design = "A", overlap = "low")
data$gen_data()
print(data$get_x()[1:2,])
```

Description

LaLonde data example

LaLonde data example

Details

Returns the LaLonde data as used by Dehja and Wahba. Note the data is fixed and `gen_data()` will just initialize the fixed data.

Value

An R6 object of class `DataSim`

Super class

`causalOT::DataSim` -> LaLonde

Methods**Public methods:**

- `LaLonde$gen_data()`
- `LaLonde$get_tau()`
- `LaLonde$gen_x()`
- `LaLonde$gen_y()`
- `LaLonde$gen_z()`
- `LaLonde$new()`
- `LaLonde$get_design()`
- `LaLonde$clone()`

Method `gen_data()`: Sets up the data

Usage:

`LaLonde$gen_data()`

Method `get_tau()`: Returns the experimental treatment effect, \$1794

Usage:

`LaLonde$get_tau()`

Method `gen_x()`: Sets up the covariate data

Usage:

`LaLonde$gen_x()`

Method `gen_y()`: Sets up the outcome data

Usage:

`LaLonde$gen_y()`

Method `gen_z()`: Sets up the treatment indicator

Usage:

`LaLonde$gen_z()`

Method `new()`: Initializes the LaLonde object.

Usage:

```
LaLonde$new(n = NULL, p = NULL, param = list(), design = "NSW", ...)
```

Arguments:

n Not used. Maintained for symmetry with other DataSim objects.

p Not used. Maintained for symmetry with other DataSim objects.

param Not used. Maintained for symmetry with other DataSim objects.

design One of "NSW" or "Full". "NSW" uses the original experimental data from the job training program while option "Full" uses the treated individuals from LaLonde's study and compares them to individuals from the Current Population Survey as controls.

... Not used.

Examples:

```
nsw <- LaLonde$new(design = "NSW")
nsw$gen_data()
nsw$get_n()
```

```
obs.study <- LaLonde$new(design = "Full")
obs.study$gen_data()
obs.study$get_n()
```

Method `get_design()`: Returns the chosen design parameters

Usage:

```
LaLonde$get_design()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
LaLonde$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
## -----
## Method `LaLonde$new`
## -----

nsw <- LaLonde$new(design = "NSW")
nsw$gen_data()
nsw$get_n()

obs.study <- LaLonde$new(design = "Full")
obs.study$gen_data()
obs.study$get_n()
```

| | |
|--------------|---|
| mean_balance | <i>Standardized absolute mean difference calculations</i> |
|--------------|---|

Description

This function will calculate the difference in means between treatment groups standardized by the pooled standard-deviation of the respective covariates.

Usage

```
mean_balance(x = NULL, z = NULL, weights = NULL, ...)
```

Arguments

| | |
|---------|--|
| x | Either a matrix, an object of class dataHolder , or an object of class DataSim |
| z | A integer vector denoting the treatments of each observations. Can be null if x is a DataSim object or already of class dataHolder . |
| weights | An object of class causalWeights . |
| ... | Not used at this time. |

Value

A vector of mean balances

Examples

```
n <- 100
p <- 6
x <- matrix(stats::rnorm(n * p), n, p)
z <- stats::rbinom(n, 1, 0.5)
weights <- calc_weight(x = x, z = z, estimand = "ATT", method = "Logistic")
mb <- mean_balance(x = x, z = z, weights = weights)
print(mb)
```

| | |
|---------|----------------|
| Measure | <i>Measure</i> |
|---------|----------------|

Description

Constructor for an R6 Measure object.

Usage

```
Measure(
  x,
  weights = NULL,
  probability.measure = TRUE,
  adapt = c("none", "weights", "x"),
  balance.functions = NA_real_,
  target.values = NA_real_,
  dtype = NULL,
  device = NULL
)
```

Arguments

| | |
|----------------------------------|---|
| <code>x</code> | The data points |
| <code>weights</code> | The empirical measure. If <code>NULL</code> , assigns equal weight to each observation |
| <code>probability.measure</code> | Is the empirical measure a probability measure? Default is <code>TRUE</code> . |
| <code>adapt</code> | Should we try to adapt the data (" <code>x</code> "), the weights (" <code>weights</code> "), or neither (" <code>none</code> "). Default is " <code>none</code> ". |
| <code>balance.functions</code> | A matrix of functions of the covariates to target for mean balance. If <code>NULL</code> and <code>target.values</code> are provided, will use the data in <code>x</code> . |
| <code>target.values</code> | The targets for the balance functions. Should be the same length as columns in <code>balance.functions</code> . |
| <code>dtype</code> | The <code>torch_tensor</code> dtype or <code>NULL</code> . |
| <code>device</code> | The device to have the data on. Should be result of <code>torch::torch_device()</code> or <code>NULL</code> . |

Details

An R6 class for representing empirical measures (data + weights) with optional gradient-based adaptation via torch.

Use `Measure()` to construct a measure. The returned object supports active bindings like `$weights` and `$x`, and methods like `$detach()`. See below for defined methods and fields.

Value

Returns a Measure object

Public fields

`balance_functions` the functions of the data that we want to adjust towards the targets
`balance_target` the values the `balance_functions` are targeting
`adapt` What aspect of the data will be adapted. One of "`none`", "`weights`", or "`x`".
`device` the `torch::torch_device()` of the data.

`dtype` the `torch::torch_dtype` of the data.

`n` the rows of the covariates, `x`.

`d` the columns of the covariates, `x`.

`probability_measure` is the measure a probability measure?

Active bindings

`grad` gets or sets gradient

`init_weights` returns the initial value of the weights

`init_data` returns the initial value of the data

`requires_grad` checks or turns on/off gradient

`weights` gets or sets weights

`x` Gets or sets the data.

Methods

Public methods:

- `Measure_$.detach()`
- `Measure_$.get_weight_parameters()`
- `Measure_$.print()`
- `Measure_$.new()`
- `Measure_$.clone()`

Method `detach()`: generates a deep clone of the object without gradients.

Usage:

`Measure_$.detach()`

Method `get_weight_parameters()`: Makes a copy of the weights parameters. prints the measure object

Usage:

`Measure_$.get_weight_parameters()`

Method `print()`:

Usage:

`Measure_$.print(...)`

Arguments:

... Not used Constructor function

Method `new()`:

Usage:

```
Measure_$new(
  x,
  weights = NULL,
  probability.measure = TRUE,
  adapt = c("none", "weights", "x"),
  balance.functions = NA_real_,
  target.values = NA_real_,
  dtype = NULL,
  device = NULL
)
```

Arguments:

`x` The data points

`weights` The empirical measure. If `NULL`, assigns equal weight to each observation

`probability.measure` Is the empirical measure a probability measure? Default is `TRUE`.

`adapt` Should we try to adapt the data ("x"), the weights ("weights"), or neither ("none"). Default is "none".

`balance.functions` A matrix of functions of the covariates to target for mean balance. If `NULL` and `target.values` are provided, will use the data in `x`.

`target.values` The targets for the balance functions. Should be the same length as columns in `balance.functions`.

`dtype` The `torch::torch_dtype` or `NULL`.

`device` The device to have the data on. Should be result of `torch::torch_device()` or `NULL`.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
Measure_$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if(torch::torch_is_installed()) {
  m <- Measure(x = matrix(0, 10, 2), adapt = "none",
              device = torch::torch_device("cpu"),
              dtype = torch::torch_double())

  print(m)
  m$x
  m$x <- matrix(1,10,2) # must have same dimensions
  m$x
  m$weights
  m$weights <- 1:10/sum(1:10)
  m$weights

  # with gradients
  m <- Measure(x = matrix(0, 10, 2),
              adapt = "weights",
              device = torch::torch_device("cpu"),
              dtype = torch::torch_double())
```

```

m$requires_grad # TRUE
m$requires_grad <- "none" # turns off
m$requires_grad # FALSE
m$requires_grad <- "x"
m$requires_grad # TRUE
m <- Measure(matrix(0, 10, 2), adapt = "none",
                device = torch::torch_device("cpu"),
                dtype = torch::torch_double())
m$grad # NULL
m <- Measure(matrix(0, 10, 2), adapt = "weights",
                device = torch::torch_device("cpu"),
                dtype = torch::torch_double())
loss <- sum(m$weights * 1:10)
loss$backward()
m$grad
# note the weights gradient is on the log softmax scale
#and the first parameter is fixed for identifiability
m$grad <- rep(1,9)
m$grad
}

```

OTProblem

OTProblem

Description

User-facing constructor for an R6 OTProblem object.

Usage

```
OTProblem(measure_1, measure_2, ...)
```

Arguments

| | |
|-----------|--|
| measure_1 | An object of class Measure |
| measure_2 | An object of class Measure |
| ... | Not used at this time |

Details

An R6 class for creating optimal transport problems with two [Measure](#) objects.

Use `OTProblem()` to construct an object of class `OTProblem`. The component objects must be of class [Measure](#).

The process of solving an OT problem involves three steps: (1) setting up the problem by creating [Measure](#) objects and combining them into an `OTProblem` object, (2) choosing the hyperparameters for the problem, and (3) solving the problem by minimizing the objective function. The first step is done by creating [Measure](#) objects and then combining them into an `OTProblem` object using the `$add()`, `$subtract()`, `$multiply()`, and `$divide()` methods. The second step is done by calling the `$setup_arguments()` method on the `OTProblem` object. The third step is done by calling the `$solve()` method on the `OTProblem` object.

Value

An R6 object of class OTProblem.

Public fields

device the `torch::torch_device()` of the data.

dtype the `torch::torch_dtype` of the data.

selected_delta the delta value selected after `choose_hyperparameters`

selected_lambda the lambda value selected after `choose_hyperparameters`

Active bindings

loss Prints the current value of the objective. Only available after the solve method has been run

penalty Returns a list of the lambda and delta penalties that will be iterated through. To set these values, use the `setup_arguments` function.

Methods**Public methods:**

- `OTProblem_$add()`
- `OTProblem_$subtract()`
- `OTProblem_$multiply()`
- `OTProblem_$divide()`
- `OTProblem_$print()`
- `OTProblem_$new()`
- `OTProblem_$setup_arguments()`
- `OTProblem_$solve()`
- `OTProblem_$choose_hyperparameters()`
- `OTProblem_$info()`
- `OTProblem_$clone()`

Method `add()`: adds o2 to the OTProblem

Usage:

```
OTProblem_$add(o2)
```

Arguments:

o2 A number or object of class OTProblem

Examples:

```
# example code
if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
```

```

y <- matrix(2, 100, 10)
m2 <- Measure(x = y,
              device = torch::torch_device("cpu"),
              dtype = torch::torch_double())

z <- matrix(3,102, 10)

m3 <- Measure(x = z,
              device = torch::torch_device("cpu"),
              dtype = torch::torch_double())

# setup OT problems
ot1 <- OTProblem(m1, m2)

ot2 <- OTProblem(m3, m2)

print(ot1)
print(ot2)

ot1$add(ot2)

print(ot1)
print(ot2)

}

```

Method `subtract()`: subtracts o2 from OTProblem

Usage:

```
OTProblem_$subtract(o2)
```

Arguments:

o2 A number or object of class OTProblem

Examples:

```

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
                device = torch::torch_device("cpu"),
                dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
                device = torch::torch_device("cpu"),
                dtype = torch::torch_double())

  z <- matrix(3,102, 10)

```

```

m3 <- Measure(x = z,
              device = torch::torch_device("cpu"),
              dtype = torch::torch_double())

# setup OT problems
ot1 <- OTProblem(m1, m2)

ot2 <- OTProblem(m3, m2)

print(ot1)
print(ot2)

ot1$subtract(ot2)

print(ot1)
print(ot2)

}

```

Method multiply(): multiplies OTProblem by o2

Usage:

```
OTProblem_$multiply(o2)
```

Arguments:

o2 A number or object of class OTProblem

Examples:

```

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3, 102, 10)

  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  # setup OT problems
  ot1 <- OTProblem(m1, m2)

  ot2 <- OTProblem(m3, m2)

```

```

print(ot1)
print(ot2)

ot1$multiply(ot2)

print(ot1)
print(ot2)

}

```

Method divide(): divides OTProblem by agument o2

Usage:

```
OTProblem_$divide(o2)
```

Arguments:

o2 A number or object of class OTProblem

Examples:

```

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3,102, 10)

  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

# setup OT problems
ot1 <- OTProblem(m1, m2)

ot2 <- OTProblem(m3, m2)

print(ot1)
print(ot2)

ot1$divide(ot2)

print(ot1)
print(ot2)

```

```
}

```

Method `print()`: prints the OT problem object

Usage:

```
OTProblem_$print(...)
```

Arguments:

... Not used at this time

Method `new()`: Constructor method

Usage:

```
OTProblem_$new(measure_1, measure_2)
```

Arguments:

`measure_1` An object of class [Measure](#)

`measure_2` An object of class [Measure](#)

... Not used at this time

Returns: An R6 object of class `OTProblem`

Method `setup_arguments()`: Sets up the OT problems for the `OTProblem` object. This should be run before `choose_hyperparameters` and `solve`.

Usage:

```
OTProblem_$setup_arguments(
  lambda,
  delta,
  grid.length = 7L,
  cost.function = NULL,
  p = 2,
  cost.online = "auto",
  debias = TRUE,
  diameter = NULL,
  ot_niter = 1000L,
  ot_tol = 0.001
)
```

Arguments:

`lambda` The penalty parameters to try for the `OTProblem`. If not provided, the function will select some.

`delta` The constraint parameters to try for the balance function problems, if any.

`grid.length` The number of hyperparameters to try if not provided

`cost.function` The cost function for the data. Can be any function that takes arguments `x1`, `x2`, `p`. Defaults to the Euclidean distance.

`p` The power to raise the cost matrix by. Default is 2

`cost.online` Should online costs be used? Default is "auto" but "tensorized" stores the cost matrix in memory while "online" will calculate it on the fly.

`debias` Should debiased a debiased `OTProblem` be used? Defaults to TRUE

diameter Diameter of the cost function.
 ot_niter Number of iterations to run the solver
 ot_tol The tolerance for convergence of the objective function

Returns: returns the object invisibly

Examples:

```
if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  ot <- OTProblem(m1, m2)
  ot$setup_arguments(lambda = 1000)
}
```

Method solve(): Solve the OTProblem at each parameter value. Must run setup_arguments first.

Usage:

```
OTProblem$solve(
  niter = 1000L,
  tol = 1e-05,
  optimizer = c("torch", "frank-wolfe"),
  torch_optim = torch::optim_lbfgs,
  torch_scheduler = torch::lr_reduce_on_plateau,
  torch_args = NULL,
  osqp_args = NULL,
  quick.balance.function = TRUE
)
```

Arguments:

niter The number of iterations to run solver at each combination of hyperparameter values
 tol The tolerance for convergence
 optimizer The optimizer to use. One of "torch" or "frank-wolfe"
 torch_optim The torch_optimizer to use. Default is [torch::optim_lbfgs](#)
 torch_scheduler The [torch::lr_scheduler](#) to use. Default is [torch::lr_reduce_on_plateau](#)
 torch_args Arguments passed to the torch optimizer and scheduler
 osqp_args Arguments passed to [osqp::osqpSettings\(\)](#) if appropriate
 quick.balance.function Should [osqp::osqp\(\)](#) be used to select balance function constraints (delta) or not. Default true.

Returns: returns the object invisibly

Examples:

```

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  ot <- OTProblem(m1, m2)
  ot$setup_arguments(lambda = 1000)
  ot$solve(niter = 1, torch_optim = torch::optim_rmsprop)
}

```

Method `choose_hyperparameters()`: Selects the hyperparameter values through a bootstrap algorithm

Usage:

```

OTProblem_$choose_hyperparameters(
  n_boot_lambda = 100L,
  n_boot_delta = 1000L,
  lambda_bootstrap = Inf
)

```

Arguments:

`n_boot_lambda` The number of bootstrap iterations to run when selecting lambda

`n_boot_delta` The number of bootstrap iterations to run when selecting delta

`lambda_bootstrap` The penalty parameter to use when selecting lambda. Higher numbers run faster.

Returns: returns the object invisibly

Examples:

```

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  ot <- OTProblem(m1, m2)
  ot$setup_arguments(lambda = c(1,1000))
  ot$solve(niter = 1, torch_optim = torch::optim_rmsprop)
  ot$choose_hyperparameters(n_boot_lambda = 2, n_boot_delta = 10, lambda_bootstrap = 100)
}

```

Method `info()`: Provides diagnostics after `solve` and `choose_hyperparameter` methods have been run.

Usage:

```
OTProblem_$info()
```

Returns: a list with slots

- `loss` the final loss values
- `iterations` The number of iterations run for each combination of parameters
- `balance.function.differences` The final differences in the balance functions
- `hyperparam.metrics` A list of the bootstrap evaluation for delta and lambda values

Examples:

```
if (torch::torch_is_installed()) {
  ot$info()
}
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
OTProblem_$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3, 100, 10)
  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  # setup OT problems
  ot1 <- OTProblem(m1, m2,
                  device = torch::torch_device("cpu"),
                  dtype = torch::torch_double())
  ot2 <- OTProblem(m3, m2,
                  device = torch::torch_device("cpu"),
                  dtype = torch::torch_double())

  # you can add or subtract OTProblem objects into
```

```

# a new OTProblem
ot <- 0.5 * ot1 + 0.5 * ot2
print(ot)

# Then you choose the hyperparameters
ot$setup_arguments(lambda = 1000)

# then you can solve the objective function
ot$solve(niter = 1, torch_optim = torch::optim_rmsprop)
}

## -----
## Method `OTProblem_$add`
## -----

# example code
if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3, 102, 10)

  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

# setup OT problems
ot1 <- OTProblem(m1, m2)

ot2 <- OTProblem(m3, m2)

print(ot1)
print(ot2)

ot1$add(ot2)

print(ot1)
print(ot2)
}

## -----
## Method `OTProblem_$subtract`
## -----

```

```

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3,102, 10)

  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  # setup OT problems
  ot1 <- OTProblem(m1, m2)

  ot2 <- OTProblem(m3, m2)

  print(ot1)
  print(ot2)

  ot1$subtract(ot2)

  print(ot1)
  print(ot2)
}

## -----
## Method `OTProblem_$multiply`
## -----

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3,102, 10)

  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),

```

```
        dtype = torch::torch_double())

# setup OT problems
ot1 <- OTProblem(m1, m2)

ot2 <- OTProblem(m3, m2)

print(ot1)
print(ot2)

ot1$multiply(ot2)

print(ot1)
print(ot2)

}

## -----
## Method `OTProblem$divide`
## -----

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  z <- matrix(3, 102, 10)

  m3 <- Measure(x = z,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  # setup OT problems
  ot1 <- OTProblem(m1, m2)

  ot2 <- OTProblem(m3, m2)

  print(ot1)
  print(ot2)

  ot1$divide(ot2)

  print(ot1)
  print(ot2)

}
```

```

## -----
## Method `OTProblem_$setup_arguments`
## -----

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  ot <- OTProblem(m1, m2)
  ot$setup_arguments(lambda = 1000)
}

## -----
## Method `OTProblem_$solve`
## -----

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())

  ot <- OTProblem(m1, m2)
  ot$setup_arguments(lambda = 1000)
  ot$solve(niter = 1, torch_optim = torch::optim_rmsprop)
}

## -----
## Method `OTProblem_$choose_hyperparameters`
## -----

if (torch::torch_is_installed()) {
  # setup measures
  x <- matrix(1, 100, 10)
  m1 <- Measure(x = x,
               device = torch::torch_device("cpu"),
               dtype = torch::torch_double())
  y <- matrix(2, 100, 10)
  m2 <- Measure(x = y, adapt = "weights",
               device = torch::torch_device("cpu"),

```

```

dtype = torch::torch_double())

ot <- OTProblem(m1, m2)
ot$setup_arguments(lambda = c(1,1000))
ot$solve(niter = 1, torch_optim = torch::optim_rmsprop)
ot$choose_hyperparameters(n_boot_lambda = 2, n_boot_delta = 10, lambda_bootstrap = 100)
}

## -----
## Method `OTProblem$info`
## -----

if (torch::torch_is_installed()) {
  ot$info()
}

```

ot_distance

Optimal Transport Distance

Description

Optimal Transport Distance

Usage

```

ot_distance(
  x1,
  x2 = NULL,
  a = NULL,
  b = NULL,
  penalty,
  p = 2,
  cost = NULL,
  debias = TRUE,
  online.cost = "auto",
  diameter = NULL,
  niter = 1000,
  tol = 1e-07
)

## S3 method for class 'causalWeights'
ot_distance(
  x1,
  x2 = NULL,
  a = NULL,
  b = NULL,
  penalty,
  p = 2,

```

```
    cost = NULL,
    debias = TRUE,
    online.cost = "auto",
    diameter = NULL,
    niter = 1000,
    tol = 1e-07
)

## S3 method for class 'matrix'
ot_distance(
  x1,
  x2,
  a = NULL,
  b = NULL,
  penalty,
  p = 2,
  cost = NULL,
  debias = TRUE,
  online.cost = "auto",
  diameter = NULL,
  niter = 1000,
  tol = 1e-07
)

## S3 method for class 'array'
ot_distance(
  x1,
  x2,
  a = NULL,
  b = NULL,
  penalty,
  p = 2,
  cost = NULL,
  debias = TRUE,
  online.cost = "auto",
  diameter = NULL,
  niter = 1000,
  tol = 1e-07
)

## S3 method for class 'torch_tensor'
ot_distance(
  x1,
  x2,
  a = NULL,
  b = NULL,
  penalty,
  p = 2,
```

```

    cost = NULL,
    debias = TRUE,
    online.cost = "auto",
    diameter = NULL,
    niter = 1000,
    tol = 1e-07
)

```

Arguments

| | |
|-------------|--|
| x1 | Either an object of class <code>causalWeights</code> or a matrix of the covariates in the first sample |
| x2 | NULL or a matrix of the covariates in the second sample. |
| a | Empirical measure of the first sample. If NULL, assumes each observation gets equal mass. Ignored for objects of class <code>causalWeights</code> . |
| b | Empirical measure of the second sample. If NULL, assumes each observation gets equal mass. Ignored for objects of class <code>causalWeights</code> . |
| penalty | The penalty of the optimal transport distance to use. If missing or NULL, the function will try to guess a suitable value depending if <code>debias</code> is TRUE or FALSE. |
| p | L_p distance metric power |
| cost | Supply your own cost function. Should take arguments x1, x2, and p. |
| debias | TRUE or FALSE. Should the debiased optimal transport distances be used. |
| online.cost | How to calculate the distance matrix. One of "auto", "tensorized", or "online". |
| diameter | The diameter of the metric space, if known. Default is NULL. |
| niter | The maximum number of iterations for the Sinkhorn updates |
| tol | The tolerance for convergence |

Value

For objects of class `matrix`, numeric value giving the optimal transport distance. For objects of class `causalWeights`, results are returned as a list for before (`'pre'`) and after adjustment (`'post'`).

Methods (by class)

- `ot_distance(causalWeights)`: method for `causalWeights` class
- `ot_distance(matrix)`: method for matrices
- `ot_distance(array)`: method for arrays
- `ot_distance(torch_tensor)`: method for `torch_tensors`

Examples

```

if ( torch::torch_is_installed() ) {
x <- matrix(stats::rnorm(10*5), 10, 5)
z <- stats::rbinom(10, 1, 0.5)
weights <- calc_weight(x = x, z = z, method = "Logistic", estimand = "ATT")
ot1 <- ot_distance(x1 = weights, penalty = 100,
p = 2, debias = TRUE, online.cost = "auto",
diameter = NULL)
ot2 <- ot_distance(x1 = x[z==0, ], x2 = x[z == 1,],
a= weights@w0/sum(weights@w0), b = weights@w1,
penalty = 100, p = 2, debias = TRUE, online.cost = "auto", diameter = NULL)

all.equal(ot1$post, ot2)
}

```

plot.causalWeights *plot.causalWeights*

Description

plot.causalWeights

Usage

```

## S3 method for class 'causalWeights'
plot(
  x,
  r_eff = NULL,
  penalty,
  p = 2,
  cost = NULL,
  debias = TRUE,
  online.cost = "auto",
  diameter = NULL,
  niter = 1000,
  tol = 1e-07,
  ...
)

```

Arguments

| | |
|---------|---|
| x | A causalWeights object |
| r_eff | The r_{eff} to use for the PSIS_diag() function. |
| penalty | The penalty of the optimal transport distance to use. If missing or NULL, the function will try to guess a suitable value depending if debias is TRUE or FALSE. |
| p | L_p distance metric power |

| | |
|-------------|---|
| cost | Supply your own cost function. Should take arguments x1, x2, and p. |
| debias | TRUE or FALSE. Should the debiased optimal transport distances be used. |
| online.cost | How to calculate the distance matrix. One of "auto", "tensorized", or "online". |
| diameter | The diameter of the metric space, if known. Default is NULL. |
| niter | The maximum number of iterations for the Sinkhorn updates |
| tol | The tolerance for convergence |
| ... | Not used at this time |

Details

The plot method first calls `summary.causalWeights` on the `causalWeights` object. Then plots the diagnostics from that summary object.

Value

The plot method returns an invisible object of class `summary_causalWeights`.

See Also

[summary.causalWeights\(\)](#)

pph

An external control trial of treatments for post-partum hemorrhage

Description

A dataset evaluating treatments for post-partum hemorrhage. The data contain treatment groups receiving misoprostol vs potential controls from other locations that received only oxytocin. The data is stored as a numeric matrix.

Usage

```
data(pph)
```

Format

A matrix with 802 rows and 17 variables

Details

The variables are as follows:

- `cum_blood_20m`. The outcome variable denoting cumulative blood loss in mL 20 minutes after the diagnosis of post-partum hemorrhage (650 – 2000).
- `tx`. The treatment indicator of whether an individual received misoprostol (1) or oxytocin (0).
- `age`. the mother's age in years (15 – 43).

- no_educ. whether a woman had no education (1) or some education (0).
- num_livebirth. the number of previous live births.
- cur_married. whether a mother is currently married (1 = yes, 0 = no).
- gest_age. the gestational age of the fetus in weeks (35 – 43).
- prev_pphys. whether the woman has had a previous post-partum hemorrhage.
- hb_test. the woman's hemoglobin in mg/dL (7 – 15).
- induced_laboryes. whether labor was induced (1 = yes, 0 = no).
- augmented_laboryes. whether labor was augmented (1 = yes, 0 = no).
- early_cordclampyes. whether the umbilical cord was clamped early (1 = yes, 0 = no).
- control_cordtractionyes. whether cord traction was controlled (1 = yes, 0 = no).
- uterine_messageyes. whether a uterine massage was given (1 = yes, 0 = no).
- placenta. whether placenta was delivered before treatment given (1 = yes, 0 = no).
- bloodlossattx. amount of blood lost when treatment given (500 mL – 1800 mL)
- sitecode. Which site is the individual from? (1 = Cairo, Egypt, 2 = Turkey, 3 = Hocmon, Vietnam, 4 = Cuchi, Vietnam, and 5 Burkina Faso).

Source

Data from the following Harvard Dataverse:

- Winikoff, Beverly, 2019, "Two randomized controlled trials of misoprostol for the treatment of postpartum hemorrhage", <https://doi.org/10.7910/DVN/ETHH4N>, Harvard Dataverse, V1.

The data was originally analyzed in

- Blum, J. et al. Treatment of post-partum haemorrhage with sublingual misoprostol versus oxytocin in women receiving prophylactic oxytocin: a double-blind, randomised, non-inferiority trial. *The Lancet* 375, 217–223 (2010).

predict.bp

Predict method for barycentric projection models

Description

Predict method for barycentric projection models

Usage

```
## S3 method for class 'bp'
predict(
  object,
  newdata = NULL,
  source.sample,
  cost_function = NULL,
  niter = 1000,
  tol = 1e-07,
  ...
)
```

Arguments

| | |
|---------------|--|
| object | An object of class "bp" |
| newdata | a data.frame containing new observations |
| source.sample | a vector giving the sample each observations arise from |
| cost_function | a cost metric between observations |
| niter | number of iterations to run the barycentric projection for powers > 2. |
| tol | Tolerance on the optimization problem for projections with powers > 2. |
| ... | Dots passed to the lbfgs method in the torch package. |

Examples

```

if(torch::torch_is_installed()) {
  set.seed(23483)
  n <- 2^5
  pp <- 6
  overlap <- "low"
  design <- "A"
  estimate <- "ATT"
  power <- 2
  data <- causalOT::Hainmueller$new(n = n, p = pp,
  design = design, overlap = overlap)

  data$gen_data()

  weights <- causalOT::calc_weight(x = data,
  z = NULL, y = NULL,
  estimand = estimate,
  method = "NNM")

  df <- data.frame(y = data$get_y(), z = data$get_z(), data$get_x())

  # unbiased
  fit <- causalOT::barycentric_projection(y ~ ., data = df,
  weight = weights,
  separate.samples.on = "z", niter = 2)

  #debiased
  fit_d <- causalOT::barycentric_projection(y ~ ., data = df,
  weight = weights,
  separate.samples.on = "z", debias = TRUE, niter = 2)

  # predictions, without new data
  unbiased_predictions <- predict(fit, source.sample = df$z)
  debiased_predictions <- predict(fit_d, source.sample = df$z)

  isTRUE(all.equal(unname(unbiased_predictions), df$y)) # FALSE
  isTRUE(all.equal(unname(debiased_predictions), df$y)) # TRUE
}

```

```
print.dataHolder      print.dataHolder
```

Description

print.dataHolder

Usage

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

Arguments

| | |
|-----|-------------------|
| x | dataHolder object |
| ... | Not used |

```
PSIS                  Pareto-Smoothed Importance Sampling
```

Description

Pareto-Smoothed Importance Sampling

Usage

```
PSIS(x, r_eff = NULL, ...)

## S4 method for signature 'numeric'
PSIS(x, r_eff = NULL, ...)

## S4 method for signature 'causalWeights'
PSIS(x, r_eff = NULL, ...)

## S4 method for signature 'list'
PSIS(x, r_eff = NULL, ...)

PSIS_diag(x, ...)

## S4 method for signature 'numeric'
PSIS_diag(x, r_eff = NULL)

## S4 method for signature 'causalWeights'
PSIS_diag(x, r_eff = NULL)
```

```
## S4 method for signature 'causalPSIS'
PSIS_diag(x, ...)

## S4 method for signature 'list'
PSIS_diag(x, r_eff = NULL)

## S4 method for signature 'psis'
PSIS_diag(x, r_eff = NULL)
```

Arguments

| | |
|--------------------|--|
| <code>x</code> | For <code>PSIS()</code> , a vector of weights, an object of class <code>causalWeights</code> , or a list with slots "w0" and "w1". For <code>PSIS_diag</code> , the results of a run of <code>PSIS()</code> . |
| <code>r_eff</code> | A vector of relative effective sample size with one estimate per observation. If providing an object of class <code>causalWeights</code> , should be a list of vectors with one vector for each sample. See <code>psis()</code> from the <code>loo</code> package for more details. Updates to the <code>loo</code> package now make it so this parameter should be ignored. |
| <code>...</code> | Arguments passed to the <code>psis()</code> function. |

Details

Acts as a wrapper to the `psis()` function from the `loo` package. It is built to handle the data types found in this package. This method is preferred to the `ESS()` function in `causalOT` since the latter is prone to error (infinite variances) but will not give good any indication that the estimates are problematic.

Value

For `PSIS()`, returns a list. See `psis()` from `loo` for a description of the outputs. Will give the log of the smoothed weights in slot `log_weights`, and in the slot `diagnostics`, it will give the `pareto_k` parameter (see the [pareto-k-diagnostic](#) page) and the `n_eff` estimates. `PSIS_diag()` returns the diagnostic slot from an object of class "psis".

Methods (by class)

- `PSIS(numeric)`: numeric weights
- `PSIS(causalWeights)`: object of class `causalWeights`
- `PSIS(list)`: list of weights
- `PSIS_diag(numeric)`: numeric weights
- `PSIS_diag(causalWeights)`: object of class `causalWeights` diagnostics
- `PSIS_diag(causalPSIS)`: diagnostics from the output of a previous call to `PSIS`
- `PSIS_diag(list)`: a list of objects
- `PSIS_diag(psis)`: output of `PSIS` function

See Also

[ESS\(\)](#)

Examples

```
x <- runif(100)
w <- x/sum(x)

res <- PSIS(x = w, r_eff = 1)
PSIS_diag(res)
```

sbwOptions

Options for the SBW method

Description

Options for the SBW method

Usage

```
sbwOptions(delta = NULL, grid.length = 20L, nboot = 1000L, ...)
```

Arguments

| | |
|-------------|--|
| delta | A number or vector of tolerances for the balancing functions. Default is NULL which will use a grid search |
| grid.length | The number of values to try in the grid search |
| nboot | The number of bootstrap samples to run during the grid search. |
| ... | Arguments passed on to osqpSettings() |

Value

A list of class sbwOptions with slots

- delta Delta values to try
- grid.length The number of parameters to try
- sumto1 Forced to be TRUE. Weights will always sum to 1.
- nboot Number of bootstrap samples
- solver.options A list with arguments passed to [osqpSettings\(\)](#)

Function balancing

This method will balance functions of the covariates within some tolerance, δ . For these functions B , we will desire

$$\frac{\sum_{i:Z_i=0} w_i B(x_i) - \sum_{j:Z_j=1} B(x_j)/n_1}{\sigma} \leq \delta$$

, where in this case we are targeting balance with the treatment group for the ATT. σ is the pooled standard deviation prior to balancing.

Examples

```
opts <- sbwOptions(delta = 0.1)
```

| | |
|------------|-----------------------------------|
| scmOptions | <i>Options for the SCM Method</i> |
|------------|-----------------------------------|

Description

Options for the SCM Method

Usage

```
scmOptions(...)
```

Arguments

... Arguments passed to the [osqpSettings\(\)](#) function which solves the problem.

Details

Options for the solver used in the optimization of the Synthetic Control Method of Abadie and Gardeazabal (2003).

Value

A list with arguments to pass to [osqpSettings\(\)](#)

Examples

```
opts <- scmOptions()
```

| | |
|-----------------------|--|
| summary.causalWeights | <i>Summary diagnostics for causalWeights</i> |
|-----------------------|--|

Description

Summary diagnostics for causalWeights

print.summary_causalWeights

plot.summary_causalWeights

Usage

```
## S3 method for class 'causalWeights'
summary(
  object,
  r_eff = NULL,
  penalty,
  p = 2,
  cost = NULL,
  debias = TRUE,
  online.cost = "auto",
  diameter = NULL,
  niter = 1000,
  tol = 1e-07,
  ...
)

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

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

Arguments

| | |
|-------------|---|
| object | an object of class causalWeights |
| r_eff | The r_eff used in the PSIS calculation. See PSIS_diag() |
| penalty | The penalty parameter to use |
| p | The power of the Lp distance to use. Overridden by argument cost. |
| cost | A user supplied cost function. Should take arguments x1, x2, p. |
| debias | Should debiased optimal transport distances be used. TRUE or FALSE |
| online.cost | Should the cost be calculated online? One of "auto", "tensorized", or "online". |
| diameter | the diameter of the covariate space. Default is NULL. |
| niter | the number of iterations to run the optimal transport distances |
| tol | the tolerance for convergence for the optimal transport distances |
| ... | Not used |
| x | an object of class "summary_causalWeights" |

Value

The summary method returns an object of class "summary_causalWeights".

Functions

- `print(summary_causalWeights)`: print method
- `plot(summary_causalWeights)`: plot method

Examples

```

if(torch::torch_is_installed()) {
  n <- 2^6
  p <- 6
  overlap <- "high"
  design <- "A"
  estimand <- "ATE"

  #### get simulation functions ####
  original <- Hainmueller$new(n = n, p = p,
                             design = design, overlap = overlap)
  original$gen_data()
  weights <- calc_weight(x = original, estimand = estimand, method = "Logistic")
  s <- summary(weights)
  plot(s)
}

```

| | |
|-------------------|--------------------------|
| supported_methods | <i>Supported Methods</i> |
|-------------------|--------------------------|

Description

Supported Methods

Usage

```
supported_methods()
```

Value

A character list with supported methods. Note "COT" is the same as "Wasserstein". We provide the second name for backwards compatibility.

Examples

```
supported_methods()
```

| | |
|-------------------|---|
| vcov.causalEffect | <i>Get the variance of a causalEffect</i> |
|-------------------|---|

Description

Get the variance of a causalEffect

Usage

```
## S3 method for class 'causalEffect'
vcov(object, ...)
```

Arguments

object An object of class `causalEffect`
... Passed on to the sandwich estimator if there is a model fit that supports one

Value

The variance of the treatment effect as a matrix

Examples

```
# set-up data
set.seed(1234)
data <- Hainmueller$new()
data$gen_data()

# calculate quantities
weight <- calc_weight(data, estimand = "ATT", method = "Logistic")
tx_eff <- estimate_effect(causalWeights = weight)

vcov(tx_eff)
```

Index

- * **datasets**
 - pph, 47
- barycentric_projection, 3
- calc_weight, 5, 20
- causalEffect, 8, 21, 56
- causalOT::DataSim, 12, 21, 24
- causalWeights, 3, 7, 19, 20, 26, 45, 46, 51, 54
- causalWeights-class, 7
- CBPS(), 6
- coef.causalEffect, 8
- cotOptions, 9
- cotOptions(), 6
- CRASH3, 12
- dataHolder, 4, 6, 14, 26
- dataHolder(), 17
- DataSim, 6, 12, 14, 15, 21, 23, 24
- df2dataHolder, 17
- entBWOptions, 18
- entBWOptions(), 6
- ESS, 19
- ESS(), 51
- ESS, causalWeights-method (ESS), 19
- ESS, numeric-method (ESS), 19
- estimate_effect, 20
- estimate_effect(), 7
- Hainmueller, 21
- LaLonde, 23
- lbfgsb3c(), 18
- mean_balance, 26
- Measure, 26, 30, 35
- Measure_ (Measure), 26
- osqp::osqp(), 36
- osqp::osqpSettings(), 36
- osqpSettings(), 52, 53
- ot_distance, 43
- OTProblem, 30
- OTProblem_ (OTProblem), 30
- pareto-k-diagnostic, 51
- plot.causalWeights, 46
- plot.summary_causalWeights
 - (summary.causalWeights), 53
- pph, 47
- predict.bp, 48
- print.dataHolder, 50
- print.summary_causalWeights
 - (summary.causalWeights), 53
- PSIS, 50
- PSIS(), 19, 20
- psis(), 51
- PSIS, causalWeights-method (PSIS), 50
- PSIS, list-method (PSIS), 50
- PSIS, numeric-method (PSIS), 50
- PSIS_diag (PSIS), 50
- PSIS_diag(), 46, 54
- PSIS_diag, causalPSIS-method (PSIS), 50
- PSIS_diag, causalWeights-method (PSIS), 50
- PSIS_diag, list-method (PSIS), 50
- PSIS_diag, numeric-method (PSIS), 50
- PSIS_diag, psis-method (PSIS), 50
- R6, 12, 15, 21, 24
- sbwOptions, 52
- sbwOptions(), 6
- scmOptions, 53
- scmOptions(), 6
- summary.causalWeights, 53
- summary.causalWeights(), 47
- supported_methods, 55
- supported_methods(), 6
- torch::lr_multiplicative(), 10

`torch::lr_reduce_on_plateau`, [36](#)
`torch::lr_scheduler`, [36](#)
`torch::optim_lbfgs`, [36](#)
`torch::optim_lbfgs()`, [10](#)
`torch::optim_rmsprop()`, [10](#)
`torch::torch_device()`, [27](#), [29](#), [31](#)
`torch::torch_dtype`, [28](#), [29](#), [31](#)
`vcov.causalEffect`, [55](#)