Package ‘CAISEr’

February 4, 2020

Type Package

Title Comparison of Algorithms with Iterative Sample Size Estimation

Version 1.0.16

Date 2020-02-03


License GPL-2

Depends R (>= 3.5.0)

Imports assertthat (>= 0.2.1), parallel (>= 3.5.1), pbmcapply (>= 1.4.1), ggplot2 (>= 3.1.1), gridExtra (>= 2.3)

Suggests MOEADr, smoof, knitr, rmarkdown, car, dplyr, pkgdown

LazyData TRUE

Encoding UTF-8

RoxygenNote 7.0.2

VignetteBuilder knitr

URL https://fcampelo.github.io/CAISEr/

BugReports https://github.com/fcampelo/CAISEr/issues

NeedsCompilation no

Author Felipe Campelo [aut, cre],
Fernanda Takahashi [ctb],
Elizabeth Wanner [ctb]

Maintainer Felipe Campelo <f.campelo@aston.ac.uk>

Repository CRAN

Date/Publication 2020-02-04 15:00:20 UTC
R topics documented:

boot_sdm ................................................................. 2
calc_instances ............................................................ 3
calc_nreps ............................................................... 6
calc_se ................................................................. 10
consolidate_partial_results ............................................ 12
dummyalgo ............................................................... 12
dummyinstance .......................................................... 14
example_SANN ............................................................ 15
calc_instances ............................................................
get_observations ......................................................... 16
plot.CAISEr .............................................................. 17
plot.nreps ............................................................... 18
plot.nreps ............................................................... 18
print.CAISEr .............................................................. 19
run_experiment ........................................................... 20
se_boot ................................................................. 25
se_param ............................................................... 26
summary.CAISEr ......................................................... 27
summary.nreps ........................................................... 29
TSP.dist ................................................................. 29

Index .................................................................. 30

<table>
<thead>
<tr>
<th>boot_sdm</th>
<th>Bootstrap the sampling distribution of the mean</th>
</tr>
</thead>
</table>

Description

Bootstraps the sampling distribution of the means for a given vector of observations

Usage

boot_sdm(x, boot.R = 999, ncpus = 1, seed = NULL)

Arguments

<table>
<thead>
<tr>
<th>x</th>
<th>vector of observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot.R</td>
<td>number of bootstrap resamples</td>
</tr>
<tr>
<td>ncpus</td>
<td>number of cores to use</td>
</tr>
<tr>
<td>seed</td>
<td>seed for the PRNG</td>
</tr>
</tbody>
</table>

Value

vector of bootstrap estimates of the sample mean
References


Author(s)
Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

Examples

```r
x <- rnorm(15, mean = 4, sd = 1)
my.sdm <- boot_sdm(x)
hist(my.sdm, breaks = 30)
qqnorm(my.sdm, pch = 20)

x <- runif(12)
my.sdm <- boot_sdm(x)
qqnorm(my.sdm, pch = 20)

# Convergence of the SDM to a Normal distribution as sample size is increased
X <- rchisq(1000, df = 3)
x1 <- rchisq(10, df = 3)
x2 <- rchisq(20, df = 3)
x3 <- rchisq(40, df = 3)
par(mfrow = c(2, 2))
plot(density(X), main = "Estimated pop distribution");
hist(boot_sdm(x1), breaks = 25, main = "SDM, n = 10")
hist(boot_sdm(x2), breaks = 25, main = "SDM, n = 20")
hist(boot_sdm(x3), breaks = 25, main = "SDM, n = 40")
par(mfrow = c(1, 1))
```

calc_instances

Calculates number of instances for the comparison of multiple algorithms

Description

Calculates either the number of instances, or the power(s) of the comparisons of multiple algorithms.

Usage

```r
calc_instances(
  ncomparisons,
  d,
  ninstances = NULL,
)```

power = NULL,
sig.level = 0.05,
alternative.side = "two.sided",
test = "t.test",
power.target = "mean"
)

Arguments

ncomparisons  number of comparisons planned
d            minimally relevant effect size (MRES, expressed as a standardized effect size,
i.e., "deviation from H0" / "standard deviation")
ninstances  the number of instances to be used in the experiment.
power           target power for the comparisons (see Details)
sig.level desired family-wise significance level (alpha) for the experiment
alternative.side type of alternative hypothesis to be performed ("two.sided" or "one.sided")
test            type of test to be used ("t.test", "wilcoxon" or "binomial")
power.target which comparison should have the desired power? Accepts "mean", "median", or "worst.case" (this last one is equivalent to the Bonferroni correction).

Details

The main use of this routine uses the closed formula of the t-test to calculate the number of instances required for the comparison of pairs of algorithms, given a desired power and standardized effect size of interest. Significance levels of each comparison are adjusted using Holm's step-down correction (the default). The routine also takes into account whether the desired statistical power refers to the mean power (the default), median, or worst-case (which is equivalent to designing the experiment for the more widely-known Bonferroni correction). See the reference by Campelo and Wanner for details.

Value

a list object containing the following items:

- ninstances - number of instances
- power - the power of the comparison
- d - the effect size
- sig.level - significance level
- alternative.side - type of alternative hypothesis
- test - type of test
Sample Sizes for Nonparametric Methods

If the parameter test is set to either Wilcoxon or Binomial, this routine approximates the number of instances using the ARE of these tests in relation to the paired t.test, using the formulas (see reference by Campelo and Takahashi for details):

\[ n_{\text{wilcox}} = n_{\text{ttest}} / 0.86 = 1.163 \times n_{\text{ttest}} \]
\[ n_{\text{binom}} = n_{\text{ttest}} / 0.637 = 1.570 \times n_{\text{ttest}} \]

Author(s)

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

References


Examples

```r
# Calculate sample size for mean-case power
K <- 10  # number of comparisons
alpha <- 0.05  # significance level
power <- 0.9  # desired power
d <- 0.5  # MRES

out <- calc_instances(K, d,
                      power = power,
                      sig.level = alpha)

# Plot power of each comparison to detect differences of magnitude d
plot(1:K, out$power,
     type = "b", pch = 20, las = 1, ylim = c(0, 1), xlab = "comparison",
     ylab = "power", xaxs = "i", xlim = c(0, K+1))
grid(NA)
points(c(0, K+1), c(power, power), type = "l", col = 2, lty = 2, lwd = .5)
text(1, 0.93, sprintf("Mean power = %2.2f for N = %d",
                        out$mean.power, out$ninstances), adj = 0)

# Check sample size if planning for Wilcoxon tests:
calc_instances(K, d,
               power = power,
               sig.level = alpha,
               test = "wilcoxon")$ninstances
```
# Calculate power profile for predefined sample size
N <- 45
out2 <- calc_instances(K, d, ninstances = N, sig.level = alpha)

points(1:K, out2$power, type = "b", pch = 19, col = 3)
text(6, .7, sprintf("Mean power = %2.2f for N = %d", out2$mean.power, out2$ninstances), adj = 0)

# Sample size for worst-case (Bonferroni) power of 0.8, using Wilcoxon
out3 <- calc_instances(K, d, power = 0.9, sig.level = alpha,
                          test = "wilcoxon", power.target = "worst.case")
out3$ninstances

# For median power:
out4 <- calc_instances(K, d, power = 0.9, sig.level = alpha,
                          test = "wilcoxon", power.target = "median")
out4$ninstances
out4$power

calc_nreps

Determine sample sizes for a set of algorithms on a single problem instance

Description

Iteratively calculates the required sample sizes for K algorithms on a given problem instance, so that the standard errors of the estimates of the pairwise differences in performance is controlled at a predefined level.

Usage

```
calc_nreps(
  instance, 
  algorithms, 
  se.max, 
  dif = "simple", 
  comparisons = "all.vs.all", 
  method = "param", 
  nstart = 20, 
  nmax = 1000, 
  seed = NULL, 
  boot.R = 499, 
  ncpus = 1, 
  force.balanced = FALSE, 
  load.folder = NA, 
  save.folder = NA
)
```
Arguments

instance  a list object containing the definitions of the problem instance. See Section Instance for details.

algorithms  a list object containing the definitions of all algorithms. See Section Algorithms for details.

se.max  desired upper limit for the standard error of the estimated difference between pairs of algorithms. See Section Pairwise Differences for details.

dif  type of difference to be used. Accepts "perc" (for percent differences) or "simple" (for simple differences)

comparisons  type of comparisons being performed. Accepts "all.vs.first" (in which cases the first object in algorithms is considered to be the reference algorithm) or "all.vs.all" (if there is no reference and all pairwise comparisons are desired).

method  method to use for estimating the standard errors. Accepts "param" (for parametric) or "boot" (for bootstrap)

nstart  initial number of algorithm runs for each algorithm. See Section Initial Number of Observations for details.

nmax  maximum total allowed number of runs to execute. Loaded results (see load.folder below) do not count towards this total.

seed  seed for the random number generator

boot.R  number of bootstrap resamples to use (if method == "boot")

ncpus  number of cores to use

force.balanced  logical flag to force the use of balanced sampling for the algorithms on each instance

load.folder  name of folder to load results from. Use either "" or "./" for the current working directory. Accepts relative paths. Use NA for not saving. calc_nreps() will look for a .RDS file with the same name

save.folder  name of folder to save the results. Use either "" or "./" for the current working directory. Accepts relative paths. Use NA for not saving.

Value

a list object containing the following items:

- instance - alias for the problem instance considered
- Xk - list of observed performance values for all algorithms
- Nk - vector of sample sizes generated for each algorithm
- Diffk - data frame with point estimates, standard errors and other information for all algorithm pairs of interest
- seed - seed used for the PRNG
- dif - type of difference used
- method - method used ("param" / "boot")
- comparisons - type of pairings ("all.vs.all" / "all.vs.first")
Instance

Parameter instance must be a named list containing all relevant parameters that define the problem instance. This list must contain at least the field instance$FUN, with the name of the function implementing the problem instance, that is, a routine that calculates y = f(x). If the instance requires additional parameters, these must also be provided as named fields.

Algorithms

Object algorithms is a list in which each component is a named list containing all relevant parameters that define an algorithm to be applied for solving the problem instance. In what follows algorithm[[k]] refers to any algorithm specified in the algorithms list.

algorithm[[k]] must contain an algorithm[[k]]$FUN field, which is a character object with the name of the function that calls the algorithm; as well as any other elements/parameters that algorithm[[k]]$FUN requires (e.g., stop criteria, operator names and parameters, etc.).

The function defined by the routine algorithm[[k]]$FUN must have the following structure: supposing that the list in algorithm[[k]] has fields algorithm[[k]]$FUN = "myalgo", algorithm[[k]]$par1 = "a" and algorithm$par2 = 5, then:

```r
myalgo <- function(par1, par2, instance, ...){
  # do stuff
  # ...
  return(results)
}
```

That is, it must be able to run if called as:

```r
# remove '$FUN' and '$alias' fields from list of arguments
# and include the problem definition as field 'instance'
myargs <- algorithm[[names(algorithm) != "FUN"]]
myargs <- myargs[names(myargs) != "alias"]
myargs$instance <- instance

# call function
do.call(algorithm$FUN,
  args = myargs)
```

The algorithm$FUN routine must return a list containing (at least) the performance value of the final solution obtained, in a field named value (e.g., result$value) after a given run.

Initial Number of Observations

In the general case the initial number of observations per algorithm (nstart) should be relatively high. For the parametric case we recommend between 10 and 20 if outliers are not expected, or between 30 and 50 if that assumption cannot be made. For the bootstrap approach we recommend using at least 20. However, if some distributional assumptions can be made - particularly low
skewness of the population of algorithm results on the test instances), then nstart can in principle be as small as 5 (if the output of the algorithms were known to be normal, it could be 1). In general, higher sample sizes are the price to pay for abandoning distributional assumptions. Use lower values of nstart with caution.

**Pairwise Differences**

Parameter dif informs the type of difference in performance to be used for the estimation ($\mu_a$ and $\mu_b$ represent the mean performance of any two algorithms on the test instance, and $\mu_\text{mu}$ represents the grand mean of all algorithms given in algorithms):

- If dif == "perc" and comparisons == "all.vs.first", the estimated quantity is $\phi_{1b} = (\mu_1 - \mu_b) / \mu_1 = 1 - (\mu_b / \mu_1)$.
- If dif == "perc" and comparisons == "all.vs.all", the estimated quantity is $\phi_{ab} = (\mu_a - \mu_b) / \mu$.
- If dif == "simple" it estimates $\mu_a - \mu_b$.

**Author(s)**

Felipe Campelo (<fcampelo@gmail.com>)

**References**


**Examples**

```r
# Example using dummy algorithms and instances. See ?dummyalgo for details.
# We generate dummy algorithms with true means 15, 10, 30, 15, 20; and true
# standard deviations 2, 4, 6, 8, 10.
algorithms <- mapply(FUN = function(i, m, s){
  list(FUN = "dummyalgo",
       alias = paste0("algo", i),
       distribution.fun = "rnorm",
       distribution.pars = list(mean = m, sd = s)),
  i = c(alg1 = 1, alg2 = 2, alg3 = 3, alg4 = 4, alg5 = 5),
  m = c(15, 10, 30, 15, 20),
  s = c(2, 4, 6, 8, 10),
})
```
SIMPLIFY = FALSE)

# Make a dummy instance with a centered (zero-mean) exponential distribution:
instance = list(FUN = "dummyinstance", distr = "rexp", rate = 5, bias = -1/5)

# Explicitate all other parameters (just this one time:
# most have reasonable default values)
myreps <- calc_nreps(instance = instance,
algorithms = algorithms,
se.max = 0.05, # desired (max) standard error
dif = "perc", # type of difference
comparisons = "all.vs.all", # differences to consider
method = "param", # method ("param", "boot")
nstart = 15, # initial number of samples
nmax = 1000, # maximum allowed sample size
seed = 1234, # seed for PRNG
boot.R = 499, # number of bootstrap resamples (unused)
ncpus = 1, # number of cores to use
force.balanced = FALSE, # force balanced sampling?
load.folder = NA, # file to load results from
save.folder = NA) # folder to save results

summary(myreps)
plot(myreps)

---
calc_se

*Calculates the standard error for simple and percent differences*

**Description**

Calculates the sample standard error for the estimator differences between multiple algorithms on a given instance.

**Usage**

```r
calc_se(Xk,
    dif = "simple",
    comparisons = "all.vs.all",
    method = "param",
    boot.R = 999)
```

**Arguments**

- **Xk**
  - list object where each position contains a vector of observations of algorithm k on a given problem instance.

- **dif**
  - name of the difference for which the SEs are desired. Accepts "perc" (for percent differences) or "simple" (for simple differences)
calc_se

comparisons standard errors to be calculated. Accepts "all.vs.first" (in which cases the first object in algorithms is considered to be the reference algorithm) or "all.vs.all" (if there is no reference and all pairwise SEs are desired).

method method used to calculate the interval. Accepts "param" (using parametric formulas based on normality of the sampling distribution of the means) or "boot" (for bootstrap).

boot.R (optional) number of bootstrap resamples (if method == "boot")

Details

- If dif == "perc" it returns the standard errors for the sample estimates of pairs \((\mu_2 - \mu_1)/\mu\), where \(\mu_1, \mu_2\) are the means of the populations that generated sample vectors \(x_1, x_2\), and
- If dif == "simple" it returns the SE for sample estimator of \((\mu_2 - \mu_1)\)

Value

a list object containing the following items:

- Phi.est - estimated values of the statistic of interest for each pair of algorithms of interest (all pairs if comparisons == "all.vs.all", or all pairs containing the first algorithm if comparisons == "all.vs.first").
- se - standard error estimates

References


Author(s)

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

Examples

# three vectors of normally distributed observations
set.seed(1234)
Xk <- list(rnorm(10, 5, 1), # mean = 5, sd = 1,
          rnorm(20, 10, 2), # mean = 10, sd = 2,
          rnorm(50, 15, 5)) # mean = 15, sd = 3

calc_se(Xk, dif = "simple", comparisons = "all.vs.all", method = "param")
calc_se(Xk, dif = "simple", comparisons = "all.vs.all", method = "boot")
calc_se(Xk, dif = "perc", comparisons = "all.vs.first", method = "param")
calc_se(Xk, dif = "perc", comparisons = "all.vs.first", method = "boot")
calc_se(Xk, dif = "perc", comparisons = "all.vs.all", method = "param")
calc_se(Xk, dif = "perc", comparisons = "all.vs.all", method = "boot")
**consolidate_partial_results**

*Consolidate results from partial files*

**Description**

Consolidates results from a set of partial files (each generated by an individual call to `calc_nreps()` into a single output structure, similar (but not identical) to the output of `run_experiment()`. This is useful e.g., to consolidate the results from interrupted experiments.

**Usage**

```r
consolidate_partial_results(Configuration, folder = "./nreps_files")
```

**Arguments**

- `Configuration`: a named list containing all parameters required in a call to `run_experiment()` except instances and algorithms. See the parameter list and default values in `run_experiment()`. Notice that this is always returned as part of the output structure of `run_experiment()`, so it generally easier to just retrieve it from previously saved results.
- `folder`: folder where the partial files are located.

**Value**

a list object containing the following fields:

- `data.raw`: data frame containing all observations generated
- `data.summary`: data frame summarizing the experiment.
- `N`: number of instances sampled
- `total.runs`: total number of algorithm runs performed
- `instances.sampled`: names of the instances sampled

---

**dummyalgo**

*Dummy algorithm routine to test the sampling procedures*

**Description**

This is a dummy algorithm routine to test the sampling procedures, in combination with `dummyinstance()`. `dummyalgo()` receives two parameters that determine the distribution of performances it will exhibit on a hypothetical problem class: `distribution.fun` (with the name of a random number generation function, e.g. `rnorm`, `runif`, `rexp` etc.); and `distribution.pars`, a named list of parameters to be passed on to `distribution.fun`. The third parameter is an instance object (see `calc_nreps()` for details), which is a named list with the following fields:
• `FUN = "dummyinstance"` - must always be "dummyinstance" (will be ignored otherwise).
• `distr` - the name of a random number generation function.
• ... - other named fields with parameters to be passed down to the function in `distr`.

### Usage
```r
dummyalgo(
  distribution.fun = "rnorm",
  distribution.pars = list(mean = 0, sd = 1),
  instance = list(FUN = "dummyinstance", distr = "rnorm", mean = 0, sd = 1)
)
```

### Arguments
- **distribution.fun**
  name of a function that generates random values according to a given distribution, e.g., "rnorm", "runif", "rexp" etc.
- **distribution.pars**
  list of named parameters required by the function in `distribution.fun`. Parameter `n` (number of points to generate) is unnecessary (this routine always forces `n = 1`).
- **instance**
  instance parameters (see Details).

### Details
`distribution.fun` and `distribution.pars` regulate the mean performance of the dummy algorithm on a given (hypothetical) problem class, and the between-instances variance of performance. The instance specification in `instance` regulates the within-instance variability of results. Ideally the distribution parameters passed to the instance should result in a within-instance distribution of values with zero mean, so that the mean of the values returned by `dummyalgo` is regulated only by `distribution.fun` and `distribution.pars`.

The value returned by `dummyalgo` is sampled as follows:

```r
  offset <- do.call(distribution.fun, args = distribution.pars)
  y   <- offset + do.call("dummyinstance", args = instance)
```

### Value
a list object with a single field `$value`, containing a scalar numerical value distributed as described at the end of Details.

### Author(s)
Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

### See Also
`dummyinstance()`
Examples

# Make a dummy instance with a centered (zero-mean) exponential distribution:
instance = list(FUN = "dummyinstance", distr = "rexp", rate = 5, bias = -1/5)

# Simulate a dummy algorithm that has a uniform distribution of expected
# performance values, between -25 and 50.
dummyalgo(distribution.fun = "runif",
           distribution.pars = list(min = -25, max = 50),
           instance = instance)

dummyinstance

Dummy instance routine to test the sampling procedures

Description

This is a dummy instance routine to test the sampling procedures, in combination with dummyalgo().
dummyinstance() receives a parameter distr containing the name of a random number generation
function (e.g. rnorm, runif, rexp etc.), plus a variable number of arguments to be passed down to
the function in distr.

Usage

dummyinstance(distr, ..., bias = 0)

Arguments

distr name of a function that generates random values according to a given distribution, e.g., "rnorm", "runif", "rexp" etc.

... additional parameters to be passed down to the function in distr. Parameter n (number of points to generate) is unnecessary (this routine always forces n = 1).

bias a bias term to add to the results of the distribution function (e.g., to set the mean to zero).

Value

a single numeric value sampled from the desired distribution.

Author(s)

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

See Also

dummyalgo()
Example 1

dummyinstance(distr = "rnorm", mean = 10, sd = 1)

# Make a centered (zero-mean) exponential distribution:
lambda = 4

# 10000 observations
set.seed(1234)
y <- numeric(10000)
for (i in 1:10000) y[i] <- dummyinstance(distr = "rexp", rate = lambda,
                                        bias = -1/lambda)

mean(y)
hist(y, breaks = 50, xlim = c(-0.5, 2.5))

---

table

Example SANN

Simulated annealing (for testing/examples)

Description

Adapted from stats::optim(). Check their documentation / examples for details.

Usage

examle_SANN(Temp, budget, instance)

Arguments

Temp controls the "SANN" method. It is the starting temperature for the cooling schedule.

budget stop criterion: number of function evaluations to execute

instance an instance object (see calc_nreps() for details)

Examples

## Not run:

instance <- list(FUN = "TSP.dist", mydist = datasets::eurodist)

example_SANN(Temp = 2000, budget = 10000, instance = instance)

## End(Not run)
get_observations

Run an algorithm on a problem.

Description

Call algorithm routine for the solution of a problem instance

Usage

get_observations(algo, instance, n = 1)

Arguments

- algo: a list object containing the definitions of the algorithm. See `calc_nreps()` for details.
- instance: a list object containing the definitions of the problem instance. See `calc_nreps()` for details.
- n: number of observations to generate.

Value

vector of observed performance values

Author(s)

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

See Also

calc_nreps()

Examples

# Make a dummy instance with a centered (zero-mean) exponential distribution:
instance <- list(FUN = "dummyinstance", distr = "rexp", rate = 5, bias = -1/5)

# Simulate a dummy algorithm that has a uniform distribution of expected
# performance values, between -25 and 50.
algorithm <- list(FUN = "dummyalgo",
distribution.fun = "runif",
distribution.pars = list(min = -25, max = 50))
x <- get_observations(algorithm, instance, n = 1000)
hist(x)
Description

S3 method for plotting CAISEr objects output by run_experiment().

Usage

```r
## S3 method for class 'CAISEr'
plot(
  x,
  y = NULL,
  ...,
  latex = FALSE,
  reorder = FALSE,
  show.text = TRUE,
  digits = 3,
  layout = NULL
)
```

Arguments

- `x` list object of class CAISEr.
- `y` unused. Included for consistency with generic plot method.
- `...` other parameters to be passed down to specific plotting functions (currently unused)
- `latex` logical: should labels be formatted for LaTeX? (useful for later saving using library TikzDevice)
- `reorder` logical: should the comparisons be reordered alphabetically?
- `show.text` logical: should text be plotted?
- `digits` how many significant digits should be used in text?
- `layout` optional parameter to override the layout of the plots (see gridExtra::arrangeGrob() for details. The default layout is `lay = rbind(c(1,1,1,1,1,1),c(1,1,1,1,1,1),c(2,2,2,3,3,3))`)

Value

list containing (1) a list of of ggplot2 objects generated, and (2) a list of data frames used for the creation of the plots.
Description

S3 method for plotting `nreps` objects output by `calc_nreps()`. 

Usage

```r
## S3 method for class 'nreps'
plot(
x, 
y = NULL,
..., 
instance.name = NULL,
latex = FALSE,
show.SE = TRUE,
show.CI = TRUE,
sig.level = 0.05,
show.text = TRUE
)
```

Arguments

- `x` list object of class `nreps` (generated by `calc_nreps()`) or of class `CAISEr` (in which case an `instance.name` must be provided).
- `y` unused. Included for consistency with generic `plot` method.
- `...` other parameters to be passed down to specific plotting functions (currently unused).
- `instance.name` name for instance to be plotted if object is of class `CAISEr`. Ignored otherwise.
- `latex` logical: should labels be formatted for LaTeX? (useful for later saving using library TikzDevice)
- `show.SE` logical: should standard errors be plotted?
- `show.CI` logical: should confidence intervals be plotted?
- `sig.level` significance level for the confidence interval. $0 < \text{sig.level} < 1$
- `show.text` logical: should text be plotted?

Value

`ggplot` object (invisibly)
Description

S3 method for printing CAISEr objects (the output of run_experiment()).

Usage

```r
## S3 method for class 'CAISEr'
print(x, ..., echo = TRUE, digits = 4, right = TRUE, breakrows = FALSE)
```

Arguments

- `x`: list object of class CAISEr (generated by run_experiment())
- `...`: other parameters to be passed down to specific summary functions (currently unused)
- `echo`: logical flag: should the print method actually print to screen?
- `digits`: the minimum number of significant digits to be used. See print.default().
- `right`: logical, indicating whether or not strings should be right-aligned.
- `breakrows`: logical, indicating whether to “widen” the output table by placing the bottom half to the right of the top half.

Value

data frame object containing the summary table (invisibly)

Examples

# Example using four dummy algorithms and 100 dummy instances.
# See [dummyalgo()] and [dummyinstance()] for details.
# Generating 4 dummy algorithms here, with means 15, 10, 30, 15 and standard
deviations 2, 4, 6, 8.
algorithms <- mapply(FUN = function(i, m, s){
  list(FUN = "dummyalgo",
       alias = paste0("algo", i),
       distribution.fun = "rnorm",
       distribution.pars = list(mean = m, sd = s)),
  i = c(alg1 = 1, alg2 = 2, alg3 = 3, alg4 = 4),
  m = c(15, 10, 30, 15),
  s = c(2, 4, 6, 8),
  SIMPLIFY = FALSE)

# Generate 100 dummy instances with centered exponential distributions
instances <- lapply(1:100, function(i) {rate <- runif(1, 1, 10)
  list(FUN = "dummyinstance",

```
run_experiment

Design and run a full experiment - calculate the required number of instances, run the algorithms on each problem instance using the iterative approach based on optimal sample size ratios, and return the results of the experiment. This routine builds upon `calc_instances()` and `calc_nreps()`, so refer to the documentation of these two functions for details.

Usage

```r
run_experiment(
  instances, algorithms,
  d, se.max,
  power = 0.8, sig.level = 0.05,
  power.target = "mean",
  dif = "simple",
  comparisons = "all.vs.all",
  alternative = "two.sided",
  test = "t.test",
  method = "param",
  nstart = 20,
  nmax = 100 * length(algorithms),
  force.balanced = FALSE,
  ncpus = 2,
  boot.R = 499,
  seed = NULL,
  save.partial.results = NA,
  load.partial.results = NA,
)```

```r
my.results <- run_experiment(instances, algorithms,
  d = 1, se.max = .1,
  power = .9, sig.level = .05,
  power.target = "mean",
  dif = "perc", comparisons = "all.vs.all",
  seed = 1234, ncpus = 1)
my.results
```
save.final.result = NA
)

Arguments

instances list object containing the definitions of the available instances. This list may (or may not) be exhausted in the experiment. To estimate the number of required instances, see `calc_instances()`. For more details, see Section Instance List.
algorithms a list object containing the definitions of all algorithms. See Section Algorithms for details.
d minimally relevant effect size (MRES), expressed as a standardized effect size, i.e., "deviation from H0" / "standard deviation". See `calc_instances()` for details.
se.max desired upper limit for the standard error of the estimated difference between pairs of algorithms. See Section Pairwise Differences for details.
power (desired) test power. See `calc_instances()` for details. Any value equal to or greater than one will force the method to use all available instances in `Instance.list`.
sig.level family-wise significance level (alpha) for the experiment. See `calc_instances()` for details.
power.target which comparison should have the desired power? Accepts "mean", "median", or "worst.case" (this last one is equivalent to the Bonferroni correction).
dif type of difference to be used. Accepts "perc" (for percent differences) or "simple" (for simple differences)
comparisons type of comparisons being performed. Accepts "all.vs.first" (in which cases the first object in `algorithms` is considered to be the reference algorithm) or "all.vs.all" (if there is no reference and all pairwise comparisons are desired).
alternative type of alternative hypothesis ("two.sided" or "less" or "greater"). See `calc_instances()` for details.
test type of test to be used ("t.test", "wilcoxon" or "binomial")
method method to use for estimating the standard errors. Accepts "param" (for parametric) or "boot" (for bootstrap)
nstart initial number of algorithm runs for each algorithm. See Section Initial Number of Observations for details.
nmax maximum number of runs to execute on each instance (see `calc_nreps()`). Loaded results (see `load.partial.results` below) do not count towards this maximum.
force.balanced logical flag to force the use of balanced sampling for the algorithms on each instance
ncpus number of cores to use
boot.R number of bootstrap resamples to use (if `method` == "boot")
seed seed for the random number generator
save.partial.results

should partial results be saved to files? Can be either NA (do not save) or a character string pointing to a folder. File names are generated based on the instance aliases. **Existing files with matching names will be overwritten.** run_experiment() uses .RDS files for saving and loading.

load.partial.results

should partial results be loaded from files? Can be either NA (do not save) or a character string pointing to a folder containing the file(s) to be loaded. run_experiment() will use .RDS file(s) with a name(s) matching instance aliases. run_experiment() uses .RDS files for saving and loading.

save.final.result

should the final results be saved to file? Can be either NA (do not save) or a character string pointing to a folder where the results will be saved on a .RDS file starting with CAISEr_results_ and ending with 12-digit datetime tag in the format YYYYMMDDhhmmss.

Value

a list object containing the following fields:

- Configuration - the full input configuration (for reproducibility)
- data.raw - data frame containing all observations generated
- data.summary - data frame summarizing the experiment.
- N - number of instances sampled
- N.star - number of instances required
- total.runs - total number of algorithm runs performed
- instances.sampled - names of the instances sampled
- Underpowered - flag: TRUE if N < N.star

Instance List

Parameter instances must contain a list of instance objects, where each field is itself a list, as defined in the documentation of function calc_nreps(). In short, each element of instances is an instance, i.e., a named list containing all relevant parameters that define the problem instance. This list must contain at least the field instance$FUN, with the name of the problem instance function, that is, a routine that calculates y = f(x). If the instance requires additional parameters, these must also be provided as named fields. An additional field, "instance$alias", can be used to provide the instance with a unique identifier (e.g., when using an instance generator).

Algorithm List

Object algorithms is a list in which each component is a named list containing all relevant parameters that define an algorithm to be applied for solving the problem instance. In what follows algorithms[[k]] refers to any algorithm specified in the algorithms list. algorithms[[k]] must contain an algorithms[[k]]$FUN field, which is a character object with the name of the function that calls the algorithm; as well as any other elements/parameters that algorithms[[k]]$FUN requires (e.g., stop criteria, operator names and parameters, etc.).
The function defined by the routine `algorithms[[k]]$FUN` must have the following structure: supposing that the list in `algorithms[[k]]` has fields `algorithm[[k]]$FUN = "myalgo", algorithms[[k]]$par1 = "a" and `algorithms[[k]]$par2 = 5`, then:

```r
myalgo <- function(par1, par2, instance, ...){
  #
  # <do stuff>
  #
  return(results)
}
```

That is, it must be able to run if called as:

```r
# remove 'FUN' and 'alias' field from list of arguments
# and include the problem definition as field 'instance'
myargs <- algorithm[names(algorithm) != "FUN"]
myargs <- myargs[names(myargs) != "alias"]
myargs$instance <- instance

# call function
do.call(algorithm$FUN,
  args = myargs)
```

The `algorithm$FUN` routine must return a list containing (at least) the performance value of the final solution obtained, in a field named `value` (e.g., `result$value`) after a given run. In general it is easier to write a small wrapper function around existing implementations.

**Initial Number of Observations**

In the general case the initial number of observations / algorithm / instance (`nstart`) should be relatively high. For the parametric case we recommend 10–15 if outliers are not expected, and 30–40 (at least) if that assumption cannot be made. For the bootstrap approach we recommend using at least 15 or 20. However, if some distributional assumptions can be made - particularly low skewness of the population of algorithm results on the test instances), then `nstart` can in principle be as small as 5 (if the output of the algorithm were known to be normal, it could be 1).

In general, higher sample sizes are the price to pay for abandoning distributional assumptions. Use lower values of `nstart` with caution.

**Pairwise Differences**

Parameter `dif` informs the type of difference in performance to be used for the estimation ($\mu_a$ and $\mu_b$ represent the mean performance of any two algorithms on the test instance, and $\mu_{\text{mu}}$ represents the grand mean of all algorithms given in `algorithms`):

- If `dif = "perc"` and `comparisons = "all.vs.first"`, the estimated quantity is: $\phi_{1b} = (\mu_1 - \mu_b)/\mu_1 = 1 - (\mu_b/\mu_1)$.
- If dif == "perc" and comparisons == "all.vs.all", the estimated quantity is: \( \phi_{ab} = (\mu_a - \mu_b)/\mu \).
- If dif == "simple" it estimates \( \mu_a - \mu_b \).

**Sample Sizes for Nonparametric Methods**

If the parameter " is set to either Wilcoxon or 'Binomial', this routine approximates the number of instances using the ARE of these tests in relation to the paired t.test:

- \( n.wilcox = n.ttest / 0.86 = 1.163 \times n.ttest \)
- \( n.binom = n.ttest / 0.637 = 1.570 \times n.ttest \)

**Author(s)**

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

**References**


**Examples**

```r
# Example using four dummy algorithms and 100 dummy instances.
# See [dummyalgo()] and [dummyinstance()] for details.
# Generating 4 dummy algorithms here, with means 15, 10, 30, 15 and standard deviations 2, 4, 6, 8.
algorithms <- mapply(FUN = function(i, m, s){
  list(FUN = "dummyalgo",
       alias = paste0("algo", i),
       distribution.fun = "rnorm",
       distribution.pars = list(mean = m, sd = s)),
  i = c(alg1 = 1, alg2 = 2, alg3 = 3, alg4 = 4),
  m = c(15, 10, 30, 15),
  s = c(2, 4, 6, 8),
  SIMPLIFY = FALSE)
```
se_boot

# Generate 100 dummy instances with centered exponential distributions
instances <- lapply(1:100,
    function(i) {rate <- runif(1, 1, 10)
        list(FUN = "dummyinstance",
            alias = paste0("Inst.", i),
            distr = "rexp", rate = rate,
            bias = -1 / rate))})

my.results <- run_experiment(instances, algorithms,
    d = .5, se.max = .1,
    power = .9, sig.level = .05,
    power.target = "mean",
    dif = "perc", comparisons = "all.vs.all",
    ncpus = 1, seed = 1234)

# Take a look at the results
summary(my.results)
plot(my.results)

se_boot

Bootstrap standard errors

Description
Calculates the standard errors of a given statistic using bootstrap

Usage
se_boot(Xk, dif = "simple", comparisons = "all.vs.all", boot.R = 999, ...)

Arguments

Xk list object where each position contains a vector of observations of algorithm k on a given problem instance.
dif name of the difference for which the SEs are desired. Accepts "perc" (for percent differences) or "simple" (for simple differences)
comparisons standard errors to be calculated. Accepts "all.vs.first" (in which cases the first object in algorithms is considered to be the reference algorithm) or "all.vs.all" (if there is no reference and all pairwise SEs are desired).
boot.R (optional) number of bootstrap resamples (if method == "boot")
... other parameters (used only for compatibility with calls to se_param(), unused in this function)

Value
Data frame containing, for each pair of interest, the estimated difference (column "Phi") and the sample standard error (column "SE")
References


Author(s)

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

Examples

# three vectors of normally distributed observations
set.seed(1234)
Xk <- list(rnorm(10, 5, 1), # mean = 5, sd = 1,
           rnorm(20, 10, 2), # mean = 10, sd = 2,
           rnorm(20, 15, 5)) # mean = 15, sd = 3

se_boot(Xk, dif = "simple", comparisons = "all.vs.all")
se_boot(Xk, dif = "perc", comparisons = "all.vs.first")
se_boot(Xk, dif = "perc", comparisons = "all.vs.all")

se_param

Parametric standard errors

Description

Calculates the standard errors of a given statistic using parametric formulas

Usage

se_param(Xk, dif = "simple", comparisons = "all.vs.all", ...)

Arguments

Xk list object where each position contains a vector of observations of algorithm k on a given problem instance.
dif name of the difference for which the SEs are desired. Accepts "perc" (for percent differences) or "simple" (for simple differences)
comparisons standard errors to be calculated. Accepts "all.vs.first" (in which cases the first object in algorithms is considered to be the reference algorithm) or "all.vs.all" (if there is no reference and all pairwise SEs are desired).
... other parameters (used only for compatibility with calls to se_boot(), unused in this function)
Value

Data frame containing, for each pair of interest, the estimated difference (column "Phi") and the sample standard error (column "SE")

References


Author(s)

Felipe Campelo (<fcampelo@ufmg.br>, <f.campelo@aston.ac.uk>)

Examples

```r
# three vectors of normally distributed observations
set.seed(1234)
Xk <- list(rnorm(10, 5, 1),  # mean = 5, sd = 1,
           rnorm(20, 10, 2), # mean = 10, sd = 2,
           rnorm(20, 15, 5)) # mean = 15, sd = 3

se_param(Xk, dif = "simple", comparisons = "all.vs.all")
se_param(Xk, dif = "perc", comparisons = "all.vs.first")
se_param(Xk, dif = "perc", comparisons = "all.vs.all")
```

Description

S3 method for summarizing CAISEr objects output by run_experiment(). Input parameters test, alternative and sig.level can be used to override the ones used in the call to run_experiment().

Usage

```r
## S3 method for class 'CAISEr'
summary(object, test = NULL, alternative = NULL, sig.level = NULL, ...)
```
Arguments

- **object**: list object of class `CAISEr` (generated by `run_experiment()`)
- **test**: type of test to be used ("t.test", "wilcoxon" or "binomial")
- **alternative**: type of alternative hypothesis ("two.sided" or "less" or "greater"). See `calc_instances()` for details.
- **sig.level**: desired family-wise significance level (alpha) for the experiment
- **...**: other parameters to be passed down to specific summary functions (currently unused)

Value

A list object is returned invisibly, containing the details of all tests performed as well as information on the total number of runs dedicated to each algorithm.

Examples

# Example using four dummy algorithms and 100 dummy instances.
# See `dummyalgo()` and `dummyinstance()` for details.
# Generating 4 dummy algorithms here, with means 15, 10, 30, 15 and standard deviations 2, 4, 6, 8.
algorithms <- mapply(FUN = function(i, m, s){
    list(FUN = "dummyalgo",
         alias = paste0("algo", i),
         distribution.fun = "rnorm",
         distribution.pars = list(mean = m, sd = s)),
    i = c(alg1 = 1, alg2 = 2, alg3 = 3, alg4 = 4),
    m = c(15, 10, 30, 15),
    s = c(2, 4, 6, 8),
    SIMPLIFY = FALSE)

# Generate 100 dummy instances with centered exponential distributions
instances <- lapply(1:100,
    function(i) {rate <- runif(1, 1, 10)
                 list(FUN = "dummyinstance",
                      alias = paste0("Inst.", i),
                      distr = "rexp", rate = rate,
                      bias = -1 / rate))}

my.results <- run_experiment(instances, algorithms,
    d = 1, se.max = .1,
    power = .9, sig.level = .05,
    power.target = "mean",
    dif = "perc", comparisons = "all.vs.all",
    seed = 1234, ncpus = 1)

summary(my.results)

# You can override some defaults if you want:
summary(my.results, test = "wilcoxon")
**summary.nreps**

**Description**

S3 method for summarizing `nreps` objects output by `calc_nreps()`.

**Usage**

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

**Arguments**

- `object` list object of class `nreps` (generated by `calc_nreps()`)
- `...` other parameters to be passed down to specific summary functions (currently unused)

**TSP.dist**

*TSP instance generator (for testing/examples)*

**Description**

Adapted from stats::optim(). Check their documentation / examples for details.

**Usage**

```r
TSP.dist(x, mydist) 
```

**Arguments**

- `x` a valid closed route for the TSP instance
- `mydist` object of class `dist` defining the TSP instance
Index

boot_sdm, 2

calc_instances, 3
calc_instances(), 20, 21, 28
calc_nreps, 6
calc_nreps(), 12, 15, 16, 18, 20–22, 29
calc_se, 10
consolidate_partial_results, 12
dummyalgo, 12
dummyalgo(), 14
dummyinstance, 14
dummyinstance(), 12, 13

eexample_SANN, 15

get_observations, 16

plot.CAISER, 17
plot.nreps, 18
print.CAISER, 19
print.default(), 19

runeperiment, 20
run_experiment(), 12, 17, 19, 27, 28

se_boot, 25
se_boot(), 26
se_param, 26
se_param(), 25
summary.CAISER, 27
summary.nreps, 29

TSP.dist, 29