Package ‘MCHT’

October 6, 2018

Title Bootstrap and Monte Carlo Hypothesis Testing

Version 0.1.0

Date 2018-10-05

Description Facilitates bootstrap and Monte Carlo hypothesis testing.

Depends R (>= 3.2)

Imports testthat, doParallel, doRNG, memoise, foreach, GenSA, purrr, stats, utils, Rdpack

RdMacros Rdpack

License MIT + file LICENSE

LazyData true

Encoding UTF-8

RoxygenNote 6.1.0

Suggests MASS, fitdistrplus

NeedsCompilation no

Author Curtis Miller [aut, cre]

Maintainer Curtis Miller <cmiller@math.utah.edu>

R topics documented:

`.onAttach` *Package Attach Hook Function*

Description

Hook triggered when package attached

Usage

```
.onAttach(lib, pkg)
```

Arguments

`lib`	a character string giving the library directory where the package defining the namespace was found
`pkg`	a character string giving the name of the package

Examples

```
MCHT:::.onAttach(.libPaths()[1], "MCHT")
```

`check_params_in_functions` *Check That Parameters Are In Functions*

Description

Test that certain parameters are arguments for certain functions via **testthat** functions.

Usage

```
check_params_in_functions(params, func_list)
```

Arguments

`params`	Character vector with parameter names
`func_list`	List of functions to check

Examples

```
MCHT:::check_params_in_functions(c("x"), list(mean))
```

gen_memo_rng *Memoised Random Variable Generation*

Description

Creates a function that generates random numbers with memoization

Usage

```
gen_memo_rng(r, seed = NULL)
```

Arguments

r	The random number generator
seed	The seed value, to be passed to `set.seed`

Details

This is a function generator, with the returned function being one that can handle a set seed and will remember if it needs to regenerate a new set of random numbers. This allows both for control over random number generation and for faster performance.

Value

A function that generates random numbers with a set seed and with memoization; accepts `seed` and all other arguments that could be passed to the original random number generator

Examples

```
memo_runif <- MCHT:::gen_memo_rng(runif)
memo_runif(10)
```

get_MCHTest_settings *Get Attributes of MCHTest Object*

Description

Get the settings of an `MCHTest`-class object.

Usage

```
get_MCHTest_settings(x)
```

Arguments

x	The `MCHTest`-class object

Value

A list with all the variables relevant to `x`

Examples

```
f <- MCHTest(mean, mean, seed = 100)
get_MCHTest_settings(f)
```

---

is.MCHTest *Is an Object of Type MCHTest?*

Description

Checks whether its argument is an `MCHTest`-class object.

Usage

```
is.MCHTest(x)
```

Arguments

`x`	An R object

Value

`TRUE` if `x` is an `MCHTest`-class objet, `FALSE` otherwise

Examples

```
f <- MCHTest(mean, mean, seed = 100)
is.MCHTest(1)
is.MCHTest(f)
```

MCHTest *Create an MCHTest Object*

Description

This function creates an `MCHTest`-class object, an S3 object that defines a bootstrap or Monte Carlo test.

Usage

```
MCHTest(test_stat, stat_gen, rand_gen = function(n) {     stats::runif(n)
  }, N = 10000, seed = NULL, memoise_sample = TRUE,
  pval_func = MCHT::pval, method = "Monte Carlo Test",
  test_params = NULL, fixed_params = NULL, nuisance_params = NULL,
  optim_control = NULL, tiebreaking = FALSE, lock_alternative = TRUE,
  threshold_pval = 1, suppress_threshold_warning = FALSE,
  localize_functions = FALSE, imported_objects = NULL)
```

Arguments

test_stat — A function that computes the test statistic from input data; x must be a parameter of this function representing test data

stat_gen — A function that generates values of the test statistic when given data; x (representing a sample) must be a parameter of this function, and this function is expected to return one numeric output, but if n is a parameter, this will be interpreted as sample size information (this could be useful for allowing a "burn-in" period in random data, as is often the case when working with time series data)

rand_gen — A function generating random data, accepting a parameter n (representing the size of the data) or x (which would be the actual data)

N — Integer representing the number of replications of stat_gen to generate

seed — The random seed used to generate simulated statistic values; if NULL, the seed will be randomly chosen each time the resulting function is called (unless memoise_sample is TRUE)

memoise_sample — If TRUE, simulated statistic values are saved and will be used repeatedly if the inputs to stat_gen don't change (such as the sample size, n); this could be in conflict with seed if seed is NULL, so set to FALSE to allow for regeneration of random samples for every call to the resulting function

pval_func — A function that computes p-values from the test statistic computed by test_stat using the simulated data generated via stat_gen; see pval for an example of how this function should be specified

method — A string labelling the test

test_params — A character vector of the names of parameters with values specified under the null hypothesis; both test_stat and stat_gen need to be able to recognize the contents of this vector as parameters (for example, if this argument is "mu", then mu needs to be an argument of both test_stat and stat_gen), and the resulting test will try to pass these parameters to rand_gen (but these *do not* need to be parameters of rand_gen)

fixed_params — A character vector of the names of parameters treated as fixed values; this isn't needed but if these parameters are being used then test output is more informative and errors will be raised if test_stat and stat_gen don't accept these parameters—which is safer—and the resulting test will try to pass these parameters to rand_gen (but these *do not* need to be parameters of rand_gen)

nuisance_params — A character vector of the names of parameters to be treated as nuisance parameters which must be chosen via optimization (see (Dufour 2006)); must be parameters of test_stat and stat_gen, but these *will not* be viewed as parameters of rand_gen, and cannot be non-NULL if codeoptim_control is NULL

optim_control — A list of arguments to be passed to GenSA, containing at least lower and upper elements as named vectors, with the names being identical to nuisance_params, but could also include other arguments to be passed to GenSA; the fn parameter will be set, and parameters of that function will be the parameters mentioned in nuisance_params, and this argument will be ignored if nuisance_params is NULL

tiebreaking — Break ties using the method as described in Dufour (2006); won't work if pval_func doesn't support it via a unif_gen argument, and should only be used for test statistics not computed on continuously-distributed data

lock_alternative
: If TRUE, then the resulting function will effectively ignore the alternative parameter, while if FALSE, the resulting function will be sensitive to values of alternative; this argument exists to prevent shooting yourself in the foot and accidentally computing p-values in inappropriate ways

threshold_pval
: A numeric value that represents a threshold p-value that, if surpassed by the optimization algorithm, will cause the algorithm to terminate; will override the threshold.stop argument in the control list that's used by GenSA

suppress_threshold_warning
: If TRUE, user will not be warned if the threshold p-value was surpassed by the optimization algorithm

localize_functions
: If TRUE, the environment of test_statss, stat_gen, rand_gen, and pval_func will be changed to the environment of the returned function; this is safer than when this is FALSE since it helps ensure there are no surprising side effects when functions in the parent environment (say, the global namespace) is changed, but if those functions depend on other functions that are not "exposed" to the resulting function (either via a parameter value or via, say, the argument imported_objects) the resulting errors can be confusing to those who don't understand how R environments work

imported_objects
: A named list of objects that will be "exposed" and localized to the environment of the returned function; this would be useful if localize_functions is TRUE but some arguments depend on other functions, because those other functions can be imported here

Details

MCHTest-class objects are effectively functions that accept data and maybe some parameters and return an htest-class object containing the results of a Monte Carlo or bootstrap statistical test. These object will accept datasets and perhaps some parameters and will return the results of a test.

Bootstrap tests can be implemented when the dataset is passed as an argument to rand_gen (which occurs when x is one of rand_gen's parameters). The only difference between a Monte Carlo test and a bootstrap test in the context of this function is that bootstrap tests use information from the original dataset when generating simulated test statistics, while a Monte Carlo test does not. When the default function for computing p-values is used, this function will perform a test similar to that described by MacKinnon (2009).

For Monte Carlo tests, when the default function for computing p-values is used (see pval), this is effectively the test described in Dufour (2006). This includes using simulated annealing to find values of nuisance parameters that maximize the p-value if the null hypothesis is true. Simulated annealing is implemented using GenSA from the **GenSA** package, and the optim_control parameter is used for controlling GenSA's behavior. We highly recommend reading GenSA's documentation.

The threshold_pval argument can be used for stopping the optimization procedure when a specified p-value is reached or surpassed. Dufour (2006) showed that p-values found using the procedure implemented here are conservative (in the sense that they are larger than they necessarily need to be). If the algorithm terminates early due to surpassing a prespecified p-value, then the estimated p-value is known to at least be the value returned, but because the p-value is a conservative estimate of the "true" p-value, this latter number could be smaller. Thus we cannot say much about the location of the true p-value if the algorithm terminates early. For this reason, a MCHTest-class function will, by default, issue a warning if the algorithm terminated early. However, by setting suppress_threshold_warning to TRUE, this behavior can be disabled. This recognizes the fact that even though an early termination leads to us not being able to say much about the location of the true p-value, we know that whatever the more accurate estimate is, we would not reject the null hypothesis based on that result.

This function uses foreach, %dorng%, and %dopar% to perform simulations. If the R session is not set up at the start for parallelization, there will be an initial complaint (after which there are no more complaints), then these functions will default to using a single core. The example shows how to set up R to use all available cores.

Due to the way environments work in R, if functions passed to test_stat, stat_gen, rand_gen, or pval_func depend on objects in the global namespace, and then those objects change, two otherwise identical runs of the resulting test could produce different results. Thus changing other objects causes side effects that may not be desired; the resulting MCHTest-class objects are no longer self-contained entities. This is why the localize_functions and imported_objects parameters exist. If localize_functions is TRUE, all of these parameters will have their environments changed to the environment in which the returned function belongs, and objects included in the list passed to imported_objects are added to this environment as well. Doing this can allow for making the MCHTest-class object self-contained and immune to side effects from changes in the global namespace. See the examples for a demonstration on how this is done. Localizing functions is safer (even though it could cause errors to be thrown when not done properly), so we highly recommend doing so.

(Beware of localizing functions from packages, like runif; if they depends on objects from the package namespace then setting localize_functions to TRUE will strip them of their package namespace and thus cause errors if they depend on other objects in that namespace. A safer approach would be to pass these objects in a wrapper function, like function(n) {runif(n)}, than passing the functions directly.)

Value

A MCHTest-class object, a function with parameters x, alternative, and ..., with other parameters being passed to functions such as those passed to test_stat and stat_gen, controlling what's tested and how; depending on lock_alternative, the alternative argument may be ignored

References

Dufour J (2006). “Monte Carlo tests with nuisance parameters: A general approach to finite-sample inference and nonstandard asymptotics.” *Journal of Econometrics*, **133**(2), 443-477. https://ideas.repec.org/a/eee/econom/v133y2006i2p443-477.html.

MacKinnon JG (2009). “Bootstrap hypothesis testing.” In Belsley DA, Kontoghiorghes EJ (eds.), *Handbook of Computational Econometrics*, 183-214. John Wiley and Sons, Ltd., West Sussex.

Examples

```
# Uncomment and run the following to set up parallelization
# library(doParallel)

# registerDoParallel(detectCores())

dat <- c(0.16, 1.00, 0.67, 1.28, 0.31, 1.16, 1.25, 0.93, 0.66, 0.54)
# Monte Carlo t-test for exponentially distributed data
mc.t.test <- MCHTest(test_stat = function(x, mu = 1) {
                       sqrt(length(x)) * (mean(x) - mu)/sd(x)
                     }, stat_gen = function(x, mu = 1) {
                       x <- x * mu
                       sqrt(length(x)) *  (mean(x) - mu)/sd(x)
```

```
                     }, rand_gen = rexp, seed = 123,
                     method = "Monte Carlo t-Test", test_params = "mu",
                     lock_alternative = FALSE)
mc.t.test(dat)
mc.t.test(dat, mu = 0.1, alternative = "two.sided")

# Testing for the scale parameter of a Weibull distribution
# Two-sided test for location of scale parameter
library(MASS)
library(fitdistrplus)

# For these examples we need to be sensitive about namespaces, or we may
# discover unwanted side effects

ts <- function(x, scale = 1) {
  fit_null <- coef(fitdist(x, "weibull", fix.arg = list("scale" = scale)))
  kt <- fit_null[["shape"]]
  l0 <- scale
  fit_all <- coef(fitdist(x, "weibull"))
  kh <- fit_all[["shape"]]
  lh <- fit_all[["scale"]]
  n <- length(x)

  # Test statistic, based on the negative-log-likelihood ratio
  suppressWarnings(n * ((kt - 1) * log(l0) - (kh - 1) * log(lh) -
      log(kt/kh) - log(lh/l0)) - (kt - kh) * sum(log(x)) + l0^(-kt) *
      sum(x^kt) - lh^(-kh) * sum(x^kh))
}

sg <- function(x, scale = 1, shape = 1) {
  x <- qweibull(x, shape = shape, scale = scale)

  # There is a reason why we're copying the original test statistic rather
  # than just calling ts() again; it has to do with environments and making
  # sure that the resulting function MCHTest() creates is independent of the
  # global namespace
  fit_null <- coef(fitdist(x, "weibull", fix.arg = list("scale" = scale)))
  kt <- fit_null[["shape"]]
  l0 <- scale
  fit_all <- coef(fitdist(x, "weibull"))
  kh <- fit_all[["shape"]]
  lh <- fit_all[["scale"]]
  n <- length(x)

  # Test statistic, based on the negative-log-likelihood ratio
  suppressWarnings(n * ((kt - 1) * log(l0) - (kh - 1) * log(lh) -
      log(kt/kh) - log(lh/l0)) - (kt - kh) * sum(log(x)) + l0^(-kt) *
      sum(x^kt) - lh^(-kh) * sum(x^kh))

  # The following would have bad side effects if ts() is redefined in the
  # global namespace
  # ts(x, scale = scale)
}

mc.wei.scale.test.1 <- MCHTest(ts, sg, seed = 123, test_params = "scale",
                               nuisance_params = "shape",
                               optim_control = list(
```

```
                               lower = c("shape" = 0),
                               upper = c("shape" = 100),
                               control = list("max.time" = 10)
                             ), threshold_pval = .2, N = 1000)

mc.wei.scale.test.1(rweibull(100, scale = 4, shape = 2), scale = 2)

# First alternative approach

sg <- function(x, scale = 1, shape = 1) {
  x <- qweibull(x, shape = shape, scale = scale)
  # The following works because test_stat will be a function in the namespace
  # of the function MCHTest() creates
  test_stat(x, scale = scale)
}

mc.wei.scale.test.2 <- MCHTest(ts, sg, seed = 123, test_params = "scale",
                             nuisance_params = "shape",
                             optim_control = list(
                               lower = c("shape" = 0),
                               upper = c("shape" = 100),
                               control = list("max.time" = 10)
                             ), threshold_pval = .2, N = 1000,
                             localize_functions = TRUE)

mc.wei.scale.test.2(rweibull(100, scale = 4, shape = 2), scale = 2)

# Second alternative approach

sg <- function(x, scale = 1, shape = 1) {
  x <- qweibull(x, shape = shape, scale = scale)
  # We will add ts() to the list of imported objects under its own name, so
  # this is now okay
  ts(x, scale = scale)
}

mc.wei.scale.test.3 <- MCHTest(ts, sg, seed = 123, test_params = "scale",
                             nuisance_params = "shape",
                             optim_control = list(
                               lower = c("shape" = 0),
                               upper = c("shape" = 100),
                               control = list("max.time" = 10)
                             ), threshold_pval = .2, N = 1000,
                             localize_functions = TRUE,
                             imported_objects = list("ts" = ts))

mc.wei.scale.test.3(rweibull(100, scale = 4, shape = 2), scale = 2)

# Bootstrap hypothesis test
# Kolmogorov-Smirnov test for Weibull distribution via parametric botstrap
# hypothesis test

ts <- function(x) {
  param <- coef(fitdist(x, "weibull"))
  shape <- param[['shape']]; scale <- param[['scale']]
  ks.test(x, pweibull, shape = shape, scale = scale,
          alternative = "two.sided")$statistic[[1]]
```

```
}

rg <- function(x) {
  n <- length(x)
  param <- coef(fitdist(x, "weibull"))
  shape <- param[['shape']]; scale <- param[['scale']]
  rweibull(n, shape = shape, scale = scale)
}

b.ks.test <- MCHTest(test_stat = ts, stat_gen = ts, rand_gen = rg,
                     seed = 123, N = 1000)
b.ks.test(rbeta(100, 2, 2))

# Permutation test

df <- data.frame(
  val = c(rnorm(5, mean = 2, sd = 3), rnorm(10, mean = 1, sd = 2)),
  group = rep(c("x", "y"), times = c(5, 10))
)

ts <- function(x) {
  means <- aggregate(val ~ group, data = x, mean)
  vars <- aggregate(val ~ group, data = x, var)
  counts <- aggregate(val ~ group, data = x, length)

  (means$val[1] - means$val[2])/sum(vars$val / sqrt(counts$val))
}

rg <- function(x) {
  x$group <- sample(x$group)
  x
}

permute.test <- MCHTest(ts, ts, rg, seed = 123, N = 1000,
                        lock_alternative = FALSE)

permute.test(df, alternative = "two.sided")
```

`MCHT_startup_message` *Create Package Startup Message*

Description

Makes package startup message.

Usage

```
MCHT_startup_message()
```

Examples

```
MCHT:::MCHT_startup_message()
```

print.MCHTest — *Print* MCHTest*-Class Object*

Description

Print an `link{MCHTest}`-class object.

Usage

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

Arguments

`x`	The `MCHTest`-class object
`...`	Other arguments, such as `prefix` (a string wrapped around the first line; by default, `"\t"`)

Examples

```
f <- MCHTest(mean, mean, seed = 100)
print(f)
```

pval — *Compute p-Value For a Test Statistic*

Description

Compute the p-value of a test statistic for Monte Carlo tests.

Usage

```
pval(S, sample_S, alternative = NULL, unif_gen = NULL)
```

Arguments

`S`	The value of the test statistic
`sample_S`	Simulated values of the
`alternative`	A string specifying the alternative hypothesis, or `NULL`
`unif_gen`	If not `NULL`, the function generating uniformly-distributed random variables for breaking ties; if `NULL`, no tie breaking is done

Details

Let S be a test statistic and S_i be simulated values of that test statistic under the null hypothesis, with $1 \leq i \leq N$. If unif_gen is not NULL, this function computes p-values via

$$p = \hat{p} = \frac{1}{N} \sum_{i=1}^{N} I_{\{(S, U_0) \leq (S_i, U_i)\}}$$

where $I_{\{S \in A\}} = 1$ if $S \in A$ and is 0 otherwise, U_i are uniformly distributed random variables, and the ordering over tuples is lexicographical ordering, as described by Dufour (2006).

If unif_gen is NULL, then the random variables are not generated and not used to break ties.

This function is designed to handle an alternative parameter similar to what appears in other **stats** functions like t.test. If alternative is "less", then $p = \hat{p}$; if alternative is "greater", then $p = 1 - \hat{p}$; and if alternative is "two.sided", then $p = 2\min(\hat{p}, 1 - \hat{p})$. Any other value raises an error.

The parameter S is S, and the vector sample_S is the vector containing the values S_i.

Value

A number representing the p-value.

References

Dufour J (2006). "Monte Carlo tests with nuisance parameters: A general approach to finite-sample inference and nonstandard asymptotics." *Journal of Econometrics*, **133**(2), 443-477. https://ideas.repec.org/a/eee/econom/v133y2006i2p443-477.html.

Examples

```
sample_S <- rnorm(10)
pval(1.01, sample_S)
pval(1.01, sample_S, alternative = "greater")
```

%s% *Concatenate (With Space)*

Description

Concatenate and form strings (with space separation)

Usage

```
x %s% y
```

Arguments

x	One object
y	Another object

Value

A string combining x and y with a space separating them

Examples

```
`%s%` <- MCHT:::`%s%`
"Hello" %s% "world"
```

%s0% *Concatenate (Without Space)*

Description

Concatenate and form strings (no space separation)

Usage

```
x %s0% y
```

Arguments

x	One object
y	Another object

Value

A string combining x and y

Examples

```
`%s0%` <- MCHT:::`%s0%`
"Hello" %s0% "world"
```

Index