R documentation

of ‘`C:/Develop/capstone/Sumit`’ etc.

August 15, 2020

R topics documented:

`calc_pca` *Performs a principle component analysis on given* matrix *or* time series *using* `RcppArmadillo`.

Description

Performs a principle component analysis on given *matrix* or *time series* using `RcppArmadillo`.

Usage

```
calc_pca(mat_rix)
```

Arguments

`mat_rix`	A *matrix* or a *time series*.

Details

The function `calc_pca()` performs a principle component analysis on a *matrix* using `RcppArmadillo`.

Value

A *matrix* of variable loadings (i.e. a matrix whose columns contain the eigenvectors).

Examples

```
## Not run:
# Create a matrix of random returns
re_turns <- matrix(rnorm(5e6), nc=5)
# Compare calc_pca() with standard prcomp()
all.equal(drop(HighFreq::calc_pca(re_turns)),
  prcomp(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=HighFreq::calc_pca(re_turns),
  rcode=prcomp(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`calc_ranksWithTies` *Calculate the ranks of the elements of a* vector *or a single-column* time series *using* `RcppArmadillo` *and* `boost`.

Description

Calculate the ranks of the elements of a *vector* or a single-column *time series* using `RcppArmadillo` and `boost`.

Usage

```
calc_ranksWithTies(vec_tor)
```

Arguments

`vec_tor`	A *vector* or a single-column *time series*.

Details

The function `calc_ranks()` calculates the ranks of the elements of a *vector* or a single-column *time series*. It *averages* the ranks in case fo ties. It uses the `boost` function `boost::sort::parallel_stable_sort` for sorting array in parallel fashion.

Value

A *double vector* with the ranks of the elements of the *vector*.

Examples

```
## Not run:
# Create a vector of random data
da_ta <- round(runif(7), 2)
# Calculate the ranks of the elements in two ways
all.equal(rank(da_ta), drop(HighFreq::calc_ranksWithTies(da_ta)))
# Create a time series of random data
da_ta <- xts::xts(runif(7), seq.Date(Sys.Date(), by=1, length.out=7))
# Calculate the ranks of the elements in two ways
all.equal(rank(coredata(da_ta)), drop(HighFreq::calc_ranksWithTies(da_ta)))
# Compare the speed of this function with RcppArmadillo and R code
da_ta <- runif(7)
library(microbenchmark)
summary(microbenchmark(
  rcpp=calc_ranks(da_ta),
  rcode=rank(da_ta),
  boost=calc_ranksWithTies(da_ta)
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`hle` *Calculate the non parametric Hodges-Lehmann estimator of location for a* vector *or a single-column* time series *using* `RcppArmadillo` *and* `RcppParallel`.

Description

Calculate the non parametric Hodges-Lehmann estimator of location for a *vector* or a single-column *time series* using `RcppArmadillo` and `RcppParallel`.

Usage

```
hle(vec_tor)
```

Arguments

`vec_tor`	A *vector* or a single-column *time series*.

Details

The function `hle()` calculates the Hodges-Lehmann estimator of the *vector*, using `RcppArmadillo` and `RcppParallel`. The function `hle()` is very much faster than function `wilcox.test()` in `R`.

Value

A single *double* value representing Hodges-Lehmann estimator of the vector.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare hle() with wilcox.test()
all.equal(drop(HighFreq::hle(re_turns)),
  wilcox.test(re_turns, conf.int = TRUE))
# Compare the speed of RcppParallel with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=HighFreq::hle(re_turns),
  rcode=wilcox.test(re_turns, conf.int = TRUE),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`KruskalWalliceTest` *Performs a Kruskal-Wallis rank sum test. using* `Rcpp` *and* `boost`*.*

Description

Performs a Kruskal-Wallis rank sum test. using `Rcpp` and `boost`.

Usage

```
KruskalWalliceTest(x)
```

Arguments

`x`	A *List* of numeric data vectors

Details

The function `KruskalWalliceTest()` performs a Kruskal-Wallis rank sum test of the null hypothesis that the location parameters of the distribution of x are the same in each group. The alternative is that they differ in at least in one.

Value

A *double* indicating p-value of the test.

Examples

```
## Not run:
x <- c(2.9, 3.0, 2.5, 2.6, 3.2) # normal subjects
y <- c(3.8, 2.7, 4.0, 2.4)      # with obstructive airway disease
z <- c(2.8, 3.4, 3.7, 2.2, 2.0) # with asbestosis

# Carry out Kruskal wallice rank sum test on the elements in two ways
all.equal(kruskal.test(list(x, y, z))$p.value, drop(HighFreq::KruskalWalliceTest(list(x, y, z))))
# Compare the speed of Rcpp and R code
library(microbenchmark)
```

```
summary(microbenchmark(
  rcpp=KruskalWalliceTest(list(x, y, z)),
  rcode=kruskal.test(list(x, y, z))$p.value,
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

---

medianAbsoluteDeviation

Calculate the Median absolute deviation of a vector *or a single-column* time series *using* RcppArmadillo.

Description

Calculate the Median absolute deviation of a *vector* or a single-column *time series* using RcppArmadillo.

Usage

```
medianAbsoluteDeviation(vec_tor)
```

Arguments

vec_tor	A *vector* or a single-column *time series*.

Details

The function `medianAbsoluteDeviation()` calculates the median of the *vector*, using RcppArmadillo. The function `medianAbsoluteDeviation()` is several times faster than `mad()` in R.

Value

A single *double* value representing median absolue deviation of the vector.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare medianAbsoluteDeviation() with mad()
all.equal(drop(HighFreq::medianAbsoluteDeviation(re_turns)),
  mad(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=HighFreq::medianAbsoluteDeviation(re_turns),
  rcode=mad(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

med_ian *Calculate the median of a* vector *or a single-column* time series *using* RcppArmadillo*.*

Description

Calculate the median of a *vector* or a single-column *time series* using RcppArmadillo.

Usage

```
med_ian(vec_tor)
```

Arguments

vec_tor	A *vector* or a single-column *time series*.

Details

The function med_ian() calculates the median of the *vector*, using RcppArmadillo. The function med_ian() is several times faster than median() in R.

Value

A single *double* value representing median of the vector.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare med_ian() with median()
all.equal(drop(HighFreq::med_ian(re_turns)),
  median(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=HighFreq::med_ian(re_turns),
  rcode=median(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`RcppArmadillo-Functions`
Set of functions in example RcppArmadillo package

Description

These four functions are created when `RcppArmadillo.package.skeleton()` is invoked to create a skeleton packages.

Usage

```
rcpparma_hello_world()
rcpparma_outerproduct(x)
rcpparma_innerproduct(x)
rcpparma_bothproducts(x)
```

Arguments

`x`	a numeric vector

Details

These are example functions which should be largely self-explanatory. Their main benefit is to demonstrate how to write a function using the Armadillo C++ classes, and to have to such a function accessible from R.

Value

`rcpparma_hello_world()` does not return a value, but displays a message to the console.

`rcpparma_outerproduct()` returns a numeric matrix computed as the outer (vector) product of `x`.

`rcpparma_innerproduct()` returns a double computer as the inner (vector) product of `x`.

`rcpparma_bothproducts()` returns a list with both the outer and inner products.

Author(s)

Dirk Eddelbuettel

References

See the documentation for Armadillo, and RcppArmadillo, for more details.

Examples

```
x <- sqrt(1:4)
rcpparma_innerproduct(x)
rcpparma_outerproduct(x)
```

`rolling_mad`	*Calculate the rolling median absolute deviation over a* vector *or a single-column* time series *using* `RcppArmadillo` *and* `RcppParallel`.

Description

Calculate the rolling median absolute deviation over a *vector* or a single-column *time series* using `RcppArmadillo` and `RcppParallel`.

Usage

```
rolling_mad(vec_tor, look_back)
```

Arguments

`vec_tor`	A *vector* or a single-column *time series*.
`look_back`	The length of look back interval, equal to the number of elements of data used for calculating the median.

Details

The function `rolling_mad()` calculates a vector of rolling medians, over a *vector* of data, using *RcppArmadillo* and *RcppParallel*.

Value

A column *vector* of the same length as the argument `vect_tor`.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
rolling_mad(re_turns)

## End(Not run)
```

`rolling_median`	*Calculate the rolling median over a* vector *or a single-column* time series *using* `RcppArmadillo` *and* `RcppParallel`.

Description

Calculate the rolling median over a *vector* or a single-column *time series* using `RcppArmadillo` and `RcppParallel`.

Usage

```
rolling_median(vec_tor, look_back)
```

Arguments

`vec_tor`	A *vector* or a single-column *time series*.
`look_back`	The length of look back interval, equal to the number of elements of data used for calculating the median.

Details

The function `rolling_median()` calculates a vector of rolling medians, over a *vector* of data, using *RcppArmadillo* and *RcppParallel*. The function `rolling_median()` is faster than `roll::roll_median()` which uses `Rcpp`.

Value

A column *vector* of the same length as the argument `vect_tor`.

Examples

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare rolling_median() with roll::roll_median()
all.equal(drop(HighFreq::rolling_median(re_turns)),
  roll::roll_median(re_turns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  parallel_rcpp=HighFreq::rolling_median(re_turns),
  rcpp=roll::roll_median(re_turns),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`TheilSenEstimator` *Calculate the non parametric Theil-Sen estimator of dependency-covariance for two* vectors *using* `RcppArmadillo`

Description

Calculate the non parametric Theil-Sen estimator of dependency-covariance for two *vectors* using `RcppArmadillo`

Usage

```
TheilSenEstimator(x, y)
```

Arguments

`vector_x`	A *vector* independent (explanatory) data.
`vector_y`	A *vector* dependent data.

Details

The function TheilSenEstimator() calculates the Theil-Sen estimator of the *vector*, using RcppArmadillo . The function TheilSenEstimator() is significantly faster than function WRS::tsreg() in R.

Value

A column *vector* containing two values i.e intercept and slope

Examples

```
## Not run:
# Create a vector of random returns
vector_x <- rnorm(10)
vactor_y <- rnorm(10)
# Compare TheilSenEstimator() with tsreg()
# Compare the speed of RcppParallel with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=HighFreq::TheilSenEstimator(vector_x, vector_y),
  rcode=WRS(vector_x, vector_y),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

WilcoxanMannWhitneyTest

Performs two sample Wilcoxan-Mann-Whitney rank sum test also known as Mann-Whitney U Test on vector *or a single-column* time series *using* RcppArmadillo *and* boost.

Description

Performs two sample Wilcoxan-Mann-Whitney rank sum test also known as Mann-Whitney U Test on *vector* or a single-column *time series* using RcppArmadillo and boost.

Usage

```
WilcoxanMannWhitneyTest(
  x,
  y,
  mu = 0,
  alternative = "two.sided",
  exact = FALSE,
  correct = TRUE
)
```

Arguments

x	A *vector* or a single-column *time series*.
y	A *vector* or a single-column *time series*.

`mu`	A *double* specifing an optional parameter used to form null hypothesis. Default value is *zero*.
`alternative`	a *character* string specifying the alternative hypothesis. It must be one of : • "two.sided" two tailed test. • "greater" greater(right) tailed test. • "less" smaller(left) tailed test. (The default is *two.sided* test.)
`exact`	A boolean indicating whether an exact p-value should be computed.
`correct`	A boolean indicating whether to apply continuity correction in normal approximation for the p-value.

Details

The function `WilcoxanMannWhitneyTest()` carries out the wilcoxan-Mann-Whitney signed rank test on *x* & *y* and returns the *p-value* of the test. By default (if `exact` is not specified), an exact p-value is computed if sample contains less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Value

A *double* indicating p-value of the test.

Examples

```
## Not run:
# Create a vector of random data
x <- round(runif(10), 2)
y <- round(runif(10), 2)
# Carry out WMW signed rank test on the elements in two ways
all.equal(wilcox.test(x, y)$p.value, drop(HighFreq::WilcoxanMannWhitneyTest(x, y)))
# Compare the speed of Rcpp and R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=WilcoxanMannWhitneyTest(x, y),
  rcode=wilcox.test(x, y),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary

## End(Not run)
```

`WilcoxanSignedRankTest`

Performs one sample Wilcoxan ranked sum test on vector *or a single-column* time series *using* `RcppArmadillo` *and* `boost`.

Description

Performs one sample Wilcoxan ranked sum test on *vector* or a single-column *time series* using `RcppArmadillo` and `boost`.

Usage

```
WilcoxanSignedRankTest(
  x,
  mu = 0,
  alternative = "two.sided",
  exact = FALSE,
  correct = TRUE
)
```

Arguments

`x`	A *vector* or a single-column *time series*.
`mu`	A *double* specifing an optional parameter used to form null hypothesis. Default value is *zero*.
`alternative`	a *character* string specifying the alternative hypothesis. It must be one of : • "two.sided" two tailed test. • "greater" greater(right) tailed test. • "less" smaller(left) tailed test. (The default is *two.sided* test.)
`exact`	A boolean indicating whether an exact p-value should be computed.
`correct`	A boolean indicating whether to apply continuity correction in normal approximation for the p-value.

Details

The function `WilcoxanSignedRankTest()` carries out the wilcoxan signed rank test on *vec_tor* and returns the *p-value* of the test. By default (if `exact` is not specified), an exact p-value is computed if sample contains less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Value

A *double* indicating p-value of the test.

Examples

```
## Not run:
# Create a vector of random data
da_ta <- round(runif(7), 2)
# Carry out wilcoxan signed rank test on the elements in two ways
all.equal(wilcox.test(da_ta)$p.value, drop(HighFreq::WilcoxanSignedRankTest(da_ta)))
# Create a time series of random data
da_ta <- xts::xts(runif(7), seq.Date(Sys.Date(), by=1, length.out=7))
# Calculate the ranks of the elements in two ways
all.equal(wilcox.test(coredata(da_ta))$p.value, drop(HighFreq::WilcoxanSignedRankTest(da_ta)))
# Compare the speed of Rcpp and R code
da_ta <- runif(10)
library(microbenchmark)
summary(microbenchmark(
  rcpp=WilcoxanSignedRankTest(da_ta),
  rcode=wilcox.test(da_ta),
  times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
## End(Not run)
```

Index