Package ‘distcomp’

October 13, 2022

Title       Computations over Distributed Data without Aggregation
Maintainer  Balasubramanian Narasimhan <naras@stat.Stanford.EDU>
Version     1.3-3
VignetteBuilder  knitr
URL         http://dx.doi.org/10.18637/jss.v077.i13
Depends     survival, stats, R (>= 3.5.0)
Imports      utils, shiny, htr (>= 1.0.0), digest, jsonlite, stringr, R6
             (>= 2.0), dplyr, rlang, magrittr, homomorpheR, gmp
Suggests    opencpu, knitr, covr, rmarkdown
Description Implementing algorithms and fitting models when sites (possibly remote) share
computation summaries rather than actual data over HTTP with a master R process (using
‘opencpu’, for example). A stratified Cox model and a singular value decomposition are
provided. The former makes direct use of code from the R ‘survival’ package. (That is,
the underlying Cox model code is derived from that in the R ‘survival’ package.)
Sites may provide data via several means: CSV files, Redcap API, etc. An extensible
design allows for new methods to be added in the future and includes facilities
for local prototyping and testing. Web applications are provided (via ‘shiny’) for
the implemented methods to help in designing and deploying the computations.

Copyright  inst/COPYRIGHTS
Encoding     UTF-8
License      LGPL (>= 2)
RoxygenNote  7.2.1
NeedsCompilation  yes
Author       Balasubramanian Narasimhan [aut, cre],
             Marina Bendersky [aut],
             Sam Gross [aut],
             Terry M. Therneau [ctb],
             Thomas Lumley [ctb]
Repository    CRAN
Date/Publication  2022-09-01 21:00:02 UTC
R topics documented:

availableComputations .................................................. 3
availableDataSources ....................................................... 4
CoxMaster ................................................................. 4
CoxWorker ................................................................. 6
createHEWorkerInstance .................................................. 7
createNCPInstance ........................................................ 8
createWorkerInstance ..................................................... 9
defineNewComputation .................................................... 10
destroyInstanceObject ..................................................... 11
distcomp ................................................................. 11
distcompSetup ............................................................ 12
executeHEMethod .......................................................... 14
executeMethod ............................................................. 14
generateId ................................................................. 15
getComputationInfo ....................................................... 15
getConfig ................................................................. 16
HEMaster ................................................................. 17
HEQueryCountMaster ..................................................... 18
HEQueryCountWorker .................................................... 20
makeDefinition ........................................................... 22
makeHEMaster ............................................................ 23
makeMaster ............................................................... 23
makeNCP ................................................................. 24
makeWorker ............................................................... 24
NCP ................................................................. 25
QueryCountMaster ....................................................... 28
QueryCountWorker ..................................................... 29
resetComputationInfo ................................................... 30
runDistcompApp .......................................................... 31
saveNewComputation ..................................................... 31
saveNewNCP ............................................................. 32
setComputationInfo ....................................................... 33
setupMaster .............................................................. 33
setupWorker ............................................................. 34
SVDMaster ............................................................... 34
SVDWorker ............................................................... 36
uploadNewComputation .................................................. 39
uploadNewNCP .......................................................... 40
writeCode ............................................................... 40

Index 42
availableComputations  

Description
The function availableComputations returns a list of available computations with various components. The names of this list (with no spaces) are unique canonical tags that are used throughout the package to unambiguously refer to the type of computation; web applications particularly rely on this list to instantiate objects. As more computations are implemented, this list is augmented.

Usage
availableComputations()

Value
a list with the components corresponding to a computation

desc a textual description (25 chars at most)
definitionApp the name of a function that will fire up a shiny webapp for defining the particular computation
workerApp the name of a function that will fire up a shiny webapp for setting up a worker site for the particular computation
masterApp the name of a function that will fire up a shiny webapp for setting up a master for the particular computation
makeDefinition the name of a function that will return a data frame with appropriate fields needed to define the particular computation assuming that they are populated in a global variable. This function is used by web applications to construct a definition object based on inputs specified by the users. Since the full information is often gathered incrementally by several web applications, the inputs are set in a global variable and therefore retrieved here using the function getComputationInfo designed for the purpose
makeMaster a function that will construct a master object for the computation given the definition and a logical flag indicating if debugging is desired
makeWorker a function that will construct a worker object for that computation given the definition and data

See Also
getComputationInfo()

Examples
availableComputations()
availableDataSources  Return currently implemented data sources

Description
The function availableDataSources returns the currently implemented data sources such as CSV files, Redcap etc.

Usage
availableDataSources()

Value
a list of named arguments, each of which is another list, with required fields named desc, a textual description and requiredPackages

Examples
availableDataSources()

CoxMaster  Create a master object to control CoxWorker worker objects

Description
CoxMaster objects instantiate and run a distributed Cox model computation fit

Methods
Public methods:
- CoxMaster$new()
- CoxMaster$kosher()
- CoxMaster$logLik()
- CoxMaster$addSite()
- CoxMaster$run()
- CoxMaster$summary()
- CoxMaster$clone()

Method new(): CoxMaster objects instantiate and run a distributed Cox model computation fit
Usage:
CoxMaster$new(defn, debug = FALSE)
Arguments:
defn a computation definition
debug a flag for debugging, default FALSE
Returns: R6 CoxMaster object

Method kosher(): Check if inputs and state of object are sane. For future use
Usage:
CoxMaster$kosher()
Returns: TRUE or FALSE

Method logLik(): Return the partial log likelihood on all data for given beta parameter.
Usage:
CoxMaster$logLik(beta)
Arguments:
beta the parameter vector
Returns: a named list with three components: value contains the value of the log likelihood, gradient contains the score vector, and hessian contains the estimated hessian matrix

Method addSite(): Add a url or worker object for a site for participating in the distributed computation. The worker object can be used to avoid complications in debugging remote calls during prototyping.
Usage:
CoxMaster$addSite(name, url = NULL, worker = NULL)
Arguments:
name of the site
url web url of the site; exactly one of url or worker should be specified
worker worker object for the site; exactly one of url or worker should be specified

Method run(): Run the distributed Cox model fit and return the estimates
Usage:
CoxMaster$run(control = coxph.control())
Arguments:
control parameters, same as survival::coxph.control()
Returns: a named list of beta, var, gradient, iter, and returnCode

Method summary():
Usage:
CoxMaster$summary()
Returns: a summary data frame columns for coef, exp(coef),’ standard error, z-score, and p-value for each parameter in the model following the same format as the survival package

Method clone(): The objects of this class are cloneable with this method.
Usage:
CoxMaster$clone(deep = FALSE)
Arguments:
deep Whether to make a deep clone.
See Also

CoxWorker which generates objects matched to such a master object

---

**CoxWorker**

* R6 class for object to use as a worker with CoxMaster master objects

---

Description

CoxWorker objects are worker objects at each data site of a distributed Cox model computation

Methods

**Public methods:**

- CoxWorker$new()
- CoxWorker$getP()
- CoxWorker$getStateful()
- CoxWorker$logLik()
- CoxWorker$var()
- CoxWorker$kosher()
- CoxWorker$clone()

**Method new():** Create a new CoxWorker object.

*Usage:*

CoxWorker$new(defn, data, stateful = TRUE)

*Arguments:*

defn the computation definition
data the local data
stateful a boolean flag indicating if state needs to be preserved between REST calls

*Returns:* a new CoxWorker object

**Method getP():** Return the dimension of the parameter vector.

*Usage:*

CoxWorker$getP(...)

*Arguments:*

... other args ignored

*Returns:* the dimension of the parameter vector

**Method getStateful():** Return the stateful status of the object.

*Usage:*

CoxWorker$getStateful()

*Returns:* the stateful flag, TRUE or FALSE
Method \texttt{logLik()}: Return the partial log likelihood on local data for given \texttt{beta} parameter.

\textit{Usage:}
\begin{verbatim}
CoxWorker$logLik(beta, ...)
\end{verbatim}

\textit{Arguments:}
- beta: the parameter vector
- ...: further arguments, currently unused

\textit{Returns:} a named list with three components: \texttt{value} contains the value of the log likelihood, \texttt{gradient} contains the score vector, and \texttt{hessian} contains the estimated hessian matrix

Method \texttt{var()}: Return the variance of estimate for given \texttt{beta} parameter on local data.

\textit{Usage:}
\begin{verbatim}
CoxWorker$var(beta, ...)
\end{verbatim}

\textit{Arguments:}
- beta: the parameter vector
- ...: further arguments, currently unused

\textit{Returns:} variance vector

Method \texttt{kosher()}: Check if inputs and state of object are sane. For future use.

\textit{Usage:}
\begin{verbatim}
CoxWorker$kosher()
\end{verbatim}

\textit{Returns:} \texttt{TRUE} or \texttt{FALSE}

Method \texttt{clone()}: The objects of this class are cloneable with this method.

\textit{Usage:}
\begin{verbatim}
CoxWorker$clone(deep = FALSE)
\end{verbatim}

\textit{Arguments:}
- deep: Whether to make a deep clone.

\textbf{See Also}

\texttt{CoxMaster} which goes hand-in-hand with this object

---

\textbf{Description}

The function \texttt{createHEWorkerInstance} uses a definition identified by \texttt{defnId} to create the appropriate object instance for HE computations. The instantiated object is searched for in the instance path and loaded if already present, otherwise it is created and assigned the instanceId and saved under the dataFileName if the latter is specified. This instantiated object may change state between iterations when a computation executes.
Usage

createHEWorkerInstance(
  defnId,
  instanceId,
  pubkey_bits = NULL,
  pubkey_n = NULL,
  den_bits = NULL,
  dataFileName = NULL
)

Arguments

defnId the identifier of an already defined computation
instanceId an identifier to use for the created instance
pubkey_bits number of bits for public key
pubkey_n the n for public key
den_bits the number of bits for the denominator
dataFileName a file name to use for saving the data. Typically NULL, this is only needed when one is using a single opencpu server to behave like multiple sites in which case the data file name serves to distinguish the site-specific data files. When it is NULL, the data file name is taken from the configuration settings

Value

TRUE if everything goes well

See Also

availableComputations()

createNCPInstance  Given the definition identifier of an object, instantiate and store object in workspace

Description

This function uses an identifier (defnId) to locate a stored definition in the workspace to create the appropriate object instance. The instantiated object is assigned the instanceId and saved under the dataFileName if the latter is not NULL. This instantiated object may change state between iterations when a computation executes
createWorkerInstance

Usage

createNCPInstance(
    name,
    ncpId,
    instanceId,
    pubkey_bits,
    pubkey_n,
    den_bits,
    dataFileName = NULL
)

Arguments

name identifying the NC party
ncpId the id indicating the NCP definition
instanceId an identifier to use for the created instance
pubkey_bits the public key number of bits
pubkey_n the pubkey n
den_bits the denominator number of bits for for rational approximations
dataFileName a file name to use for saving the data. Typically NULL, this is only needed when one is using a single opencpu server to behave like multiple sites in which case the data file name serves to distinguish the site-specific data files. When it is NULL, the data file name is taken from the configuration settings.

Value

TRUE if everything goes well

description

The function createWorkerInstance uses a definition identified by defnId to create the appropriate object instance. The instantiated object is assigned the instanceId and saved under the dataFileName if the latter is specified. This instantiated object may change state between iterations when a computation executes.
defineNewComputation

Usage

createWorkerInstance(
    defnId,
    instanceId,
    pubkey_bits = NULL,
    pubkey_n = NULL,
    den_bits = NULL,
    dataFileName = NULL
)

Arguments

    defnId       the identifier of an already defined computation
    instanceId   an identifier to use for the created instance
    pubkey_bits  number of bits for public key
    pubkey_n     the n for public key
    den_bits     the number of bits for the denominator
    dataFileName a file name to use for saving the data. Typically NULL, this is only needed when
                  one is using a single opencpu server to behave like multiple sites in which case
                  the data file name serves to distinguish the site-specific data files. When it is
                  NULL, the data file name is taken from the configuration settings

Value

    TRUE if everything goes well

See Also

    availableComputations()
**destroyInstanceObject**  
*Destroy an instance object given its identifier*

**Description**

The function `destroyInstanceObject` deletes an object associated with the `instanceId`. This is typically done after a computation completes and results have been obtained.

**Usage**

`destroyInstanceObject(instanceId)`

**Arguments**

- `instanceId`  
  the id of the object to destroy

**Value**

TRUE if everything goes well

**See Also**

`createWorkerInstance()`

---

**distcomp**  
*Distributed Computing with R*

**Description**

`distcomp` is a collection of methods to fit models to data that may be distributed at various sites. The package arose as a way of addressing the issues regarding data aggregation; by allowing sites to have control over local data and transmitting only summaries, some privacy controls can be maintained. Even when participants have no objections in principle to data aggregation, it may still be useful to keep data local and expose just the computations. For further details, please see the reference cited below.

**Details**

The initial implementation consists of a stratified Cox model fit with distributed survival data and a Singular Value Decomposition of a distributed matrix. General Linear Models will soon be added. Although some sanity checks and balances are present, many more are needed to make this truly robust. We also hope that other methods will be added by users.

We make the following assumptions in the implementation: (a) the aggregate data is logically a stacking of data at each site, i.e., the full data is row-partitioned into sites where the rows are observations; (b) Each site has the package `distcomp` installed and a workspace setup for (writeable)
use by the opencpu server (see `distcompSetup()`, and (c) each site is exposing `distcomp` via an opencpu server.

The main computation happens via a master process, a script of R code, that makes calls to `distcomp` functions at worker sites via opencpu. The use of opencpu allows developers to prototype their distributed implementations on a local machine using the opencpu package that runs such a server locally using localhost ports.

Note that `distcomp` computations are not intended for speed/efficiency; indeed, they are orders of magnitude slower. However, the models that are fit are not meant to be recomputed often. These and other details are discussed in the paper mentioned above.

The current implementation, particularly the Stratified Cox Model, makes direct use of code from `survival::coxph()`. That is, the underlying Cox model code is derived from that in the R `survival` survival package.

For an understanding of how this package is meant to be used, please see the documented examples and the reference.

References


See Also

The examples in `system.file("doc", "examples.html", package="distcomp")`

The source for the examples: `system.file("doc_src", "examples.Rmd", package="distcomp")`.

---

**distcompSetup**

Setup a workspace and configuration for a distributed computation

**Description**

The function `distcompSetup` sets up a distributed computation and configures some global parameters such as definition file names, data file names, instance object file names, and ssl configuration parameters. The function creates some of necessary subdirectories if not already present and throws an error if the workspace areas are not writeable.

**Usage**

```r
distcompSetup(
  workspacePath = "",
  defnPath = paste(workspacePath, "defn", sep = .Platform$file.sep),
  instancePath = paste(workspacePath, "instances", sep = .Platform$file.sep),
  defnFileName = "defn.rds",
  dataFileName = "data.rds",
)```
instanceFileName = "instance.rds",
resultsCacheFile = "results_cache.rds",
ssl_verifyhost = 1L,
ssl_verifypeer = 1L
)

Arguments

workspacePath a folder specifying the workspace path. This has to be writable by the opencpu process. On a cloud opencpu server on Ubuntu, for example, this requires a one-time modification of apparmor profiles to enable write permissions to this path
defnPath the path where definition files will reside, organized by computation identifiers
instancePath the path where instance objects will reside
defnFileName the name for the compdef definition files
dataFileName the name for the data files
instanceFileName the name for the instance files
resultsCacheFileName the name for the instance results cache files for HE computations
ssl_verifyhost integer value, usually 1L, but for testing with snake-oil certs, one might set this to 0L
ssl_verifypeer integer value, usually 1L, but for testing with snake-oil certs, one might set this to 0L

Value

TRUE if all is well

See Also

getConfig()

Examples

## Not run:
distcompSetup(workspacePath="/workspace")

## End(Not run)
executeHEMethod

Given the id of a serialized object, invoke a method on the object with arguments using homomorphic encryption

Description

The function executeHEMethod is a homomorphic encryption wrapper around executeMethod. It ensures any returned result is encrypted using the homomorphic encryption function.

Usage

executeHEMethod(objectId, method, ...)

Arguments

objectId the (instance) identifier of the object on which to invoke a method
method the name of the method to invoke
... further arguments as appropriate for the method

Value

a list containing an integer and a fractional result converted to characters

executeMethod

Given the id of a serialized object, invoke a method on the object with arguments

Description

The function executeMethod is really the heart of distcomp. It executes an arbitrary method on an object that has been serialized to the distcomp workspace with any specified arguments. The result, which is dependent on the computation that is executed, is returned. If the object needs to save state between iterations on it, it is automatically serialized back for the ensuing iterations.

Usage

executeMethod(objectId, method, ...)

Arguments

objectId the (instance) identifier of the object on which to invoke a method
method the name of the method to invoke
... further arguments as appropriate for the method

Value

a result that depends on the computation being executed
**generateId**

*Generate an identifier for an object*

**Description**

A hash is generated based on the contents of the object

**Usage**

```r
generateId(object, algo = "xxhash64")
```

**Arguments**

- `object` the object for which a hash is desired
- `algo` the algorithm to use, default is "xxhash64" from `digest::digest()`

**Value**

the hash as a string

**See Also**

`digest::digest()`

---

**getComputationInfo**

*Get the value of a variable from the global store*

**Description**

In distcomp, several web applications need to communicate between themselves. Since only one application is expected to be active at any time, they do so via a global store, essentially a hash table. This function retrieves the value of a name

**Usage**

```r
getComputationInfo(name)
```

**Arguments**

- `name` the name for the object

**Value**

the value for the variable, NULL if not set

**See Also**

`setComputationInfo()`
**getConfig**

*Return the workspace and configuration setup values*

**Description**

The function `getConfig` returns the values of the configuration parameters set up by `distcompSetup`

**Usage**

`getConfig(...)`

**Arguments**

... any further arguments

**Value**

A list consisting of:

- `workspacePath` a folder specifying the workspace path. This has to be writable by the opencpu process. On a cloud opencpu server on Ubuntu, for example, this requires a one-time modification of apparmor profiles to enable write permissions to this path
- `defnPath` the path where definition files will reside, organized by computation identifiers
- `instancePath` the path where instance objects will reside
- `defnFileName` the name for the compdef definition files
- `dataFileName` the name for the data files
- `instanceFileName` the name for the instance files
- `ssl_verifyhost` integer value, usually 1L, but for testing with snake-oil certs, one might set this to 0L
- `ssl_verifypeer` integer value, usually 1L, but for testing with snake-oil certs, one might set this to 0L

**See Also**

`distcompSetup()`

**Examples**

```r
## Not run:
getConfig()

## End(Not run)
```
HEMaster

Create a HEMaster process for use in a distributed homomorphic encrypted (HE) computation

Description

HEMaster objects run a distributed computation based upon a definition file that encapsulates all information necessary to perform a computation. A master makes use of two non-cooperating parties which communicate with sites that perform the actual computations using local data.

Public fields

den denominator for rational arithmetic
den_bits number of bits for denominator for rational arithmetic

Methods

Public methods:

- HEMaster$new()
- HEMaster$getNC_party()
- HEMaster$getPubkey()
- HEMaster$addNC()
- HEMaster$run()
- HEMaster$clone()

Method new(): Create a HEMaster object to run homomorphic encrypted computation

Usage:
HEMaster$new(defn)

Arguments:
defn the homomorphic computation definition

Returns: a HEMaster object

Method getNC_party(): Return a list of noncooperating parties (NCPs)

Usage:
HEMaster$getNC_party()

Returns: a named list of length 2 of noncooperating party information

Method getPubkey(): Return the public key from the public private key pair

Usage:
HEMaster$getPubkey()

Returns: an R6 Pubkey object

Method addNC(): Add a noncooperating party to this master either using a url or an object in session for prototyping
Usage:
HEMaster$addNCP(ncp_defn, url = NULL, ncpWorker = NULL)

Arguments:
ncp_defn the definition of the NCP
url the url for the NCP; only one of url and ncpWorker should be non-null
ncpWorker an instantiated worker object; only one of url and ncpWorker should be non-null

Method run(): Run a distributed homomorphic encrypted computation and return the result

Usage:
HEMaster$run(debug = FALSE)

Arguments:
debug a flag for debugging, default FALSE

Returns: the result of the distributed homomorphic computation

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

Usage:
HEMaster$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also
NCP()
**Methods**

**Public methods:**

- `HEQueryCountMaster$new()`
- `HEQueryCountMaster$setParams()`
- `HEQueryCountMaster$kosher()`
- `HEQueryCountMaster$queryCount()`
- `HEQueryCountMaster$run()`
- `HEQueryCountMaster$cleanup()`
- `HEQueryCountMaster$clone()`

**Method `new()`**: Create a new `HEQueryCountMaster` object.

*Usage:*

`HEQueryCountMaster$new(defn, partyNumber, debug = FALSE)`

*Arguments:*

- `defn` the computation definition
- `partyNumber` the party number of the NCP that this object belongs to (1 or 2)
- `debug` a flag for debugging, default FALSE

*Returns:* a new `HEQueryCountMaster` object

**Method `setParams()`**: Set some parameters of the `HEQueryCountMaster` object for homomorphic computations

*Usage:*

`HEQueryCountMaster$setParams(pubkey_bits, pubkey_n, den_bits)`

*Arguments:*

- `pubkey_bits` the number of bits in public key
- `pubkey_n` the n for the public key
- `den_bits` the number of bits in the denominator (power of 2) used in rational approximations

**Method `kosher()`**: Check if inputs and state of object are sane. For future use

*Usage:*

`HEQueryCountMaster$kosher()`

*Returns:* TRUE or FALSE

**Method `queryCount()`**: Run the distributed query count, associate it with a token, and return the result

*Usage:*

`HEQueryCountMaster$queryCount(token)`

*Arguments:*

- `token` a token to use as key

*Returns:* the partial result as a list of encrypted items with components int and frac

**Method `cleanup()`**: Cleanup the instance objects
**Usage:**
HEQueryCountMaster\$cleanup()

**Method** run()

Usage:
HEQueryCountMaster\$run(token)

Arguments:
token a token to use as key

Returns: the partial result as a list of encrypted items with components int and frac

**Method** clone()

Usage:
HEQueryCountMaster\$clone(deep = FALSE)

Arguments:
dee p Whether to make a deep clone.

**See Also**

HEQueryCountWorker() which goes hand-in-hand with this object

---

HEQueryCountWorker Create a homomorphic computation query count worker object for use with master objects generated by HEQueryCountMaster()

**Description**

HEQueryCountWorker objects are worker objects at each site of a distributed query count model computation using homomorphic encryption

**Super class**

distcomp::QueryCountWorker -> HEQueryCountWorker

**Public fields**

pub key the master’s public key visible to everyone
den the denominator for rational arithmetic
Methods

Public methods:
- HEQueryCountWorker$new()
- HEQueryCountWorker$setParams()
- HEQueryCountWorker$queryCount()
- HEQueryCountWorker$clone()

Method new(): Create a new HEQueryMaster object.
Usage:
```
HEQueryCountWorker$new(
  defn,
  data,
  pubkey_bits = NULL,
  pubkey_n = NULL,
  den_bits = NULL
)
```
Arguments:
defn the computation definition
data the data which is usually the list of sites
pubkey_bits the number of bits in public key
pubkey_n the n for the public key
den_bits the number of bits in the denominator (power of 2) used in rational approximations
Returns: a new HEQueryMaster object

Method setParams(): Set some parameters for homomorphic computations
Usage:
```
HEQueryCountWorker$setParams(pubkey_bits, pubkey_n, den_bits)
```
Arguments:
pubkey_bits the number of bits in public key
pubkey_n the n for the public key
den_bits the number of bits in the denominator (power of 2) used in rational approximations

Method queryCount(): Run the query count on local data and return the appropriate encrypted result to the party
Usage:
```
HEQueryCountWorker$queryCount(partyNumber, token)
```
Arguments:
partyNumber the NCP party number (1 or 2)
token a token to use for identifying parts of the same computation for NCP1 and NCP2
Returns: the count as a list of encrypted items with components int and frac

Method clone(): The objects of this class are cloneable with this method.
Usage:
```
HEQueryCountWorker$clone(deep = FALSE)
```
Arguments:
deep Whether to make a deep clone.
See Also

HEQueryCountMaster() which goes hand-in-hand with this object

| makeDefinition | Make a computation definition given the computation type |

Description

The function `makeDefinition` returns a computational definition based on current inputs (from the global store) given a canonical computation type tag. This is a utility function for web applications to use as input is being gathered.

Usage

`makeDefinition(compType)`

Arguments

- `compType`: the canonical computation type tag

Value

a data frame corresponding to the computation type

See Also

`availableComputations()`

Examples

```r
## Not run:
makeDefinition(names(availableComputations())[1])
## End(Not run)
```
**makeHEMaster**

**Description**

Instantiate a master process for HE operations

**Usage**

`makeHEMaster(defn)`

**Arguments**

- `defn` the computation definition

**Value**

an master object for HE operations

**makeMaster**

**Description**

The function `makeMaster` returns a master object corresponding to the definition. The types of master objects that can be created depend upon the available computations

**Usage**

`makeMaster(defn, partyNumber = NULL, debug = FALSE)`

**Arguments**

- `defn` the computation definition
- `partyNumber` the number of the noncooperating party, which can be optionally set if HE is desired
- `debug` a debug flag

**Value**

a master object of the appropriate class based on the definition

**See Also**

`availableComputations()`
makeNCP

*Instantiate an noncooperating party*

**Description**

Instantiate an noncooperating party

**Usage**

```r
makeNCP(
  ncp_defn, comp_defn,
  sites = list(),
  pubkey_bits = NULL,
  pubkey_n = NULL,
  den_bits = NULL
)
```

**Arguments**

- `ncp_defn`: the NCP definition
- `comp_defn`: the computation definition
- `sites`: a list of sites each entry a named list of name, url, worker
- `pubkey_bits`: number of bits for public key
- `pubkey_n`: the n for the public key
- `den_bits`: the log to base 2 of the denominator

**Value**

an NCP object

---

makeWorker

*Make a worker object given a definition and data*

**Description**

The function `makeWorker` returns an object of the appropriate type based on a computation definition and sets the data for the object. The types of objects that can be created depend upon the available computations

**Usage**

```r
makeWorker(defn, data, pubkey_bits = NULL, pubkey_n = NULL, den_bits = NULL)
```
Arguments

defn the computation definition
data the data for the computation
pubkey_bits the number of bits for the public key (used only if he is TRUE in computation definition)
pubkey_n the n for public key (used only if he is TRUE in computation definition)
den_bits the number of bits for the denominator (used only if he is TRUE in computation definition)

Value

a worker object of the appropriate class based on the definition

See Also

availableComputations()

NCP R6 object to use as non-cooperating party in a distributed homomorphic computation

Description

NCP objects are worker objects that separate a master process from communicating directly with the worker processes. Typically two such are needed for a distributed homomorphic computation. A master process can communicate with NCP objects and the NCP objects can communicate with worker processes. However, the two NCP objects, designated by numbers 1 and 2, are non-cooperating in the sense that they don’t communicate with each other and are isolated from each other.

Public fields

pubkey the master’s public key visible to everyone
pubkey_bits the number of bits in the public key (used for reconstructing public key remotely by serializing to character)
pubkey_n the n for the public key used for reconstructing public key remotely
den the denominator for rational arithmetic
den_bits the number of bits in the denominator used for reconstructing denominator remotely
Methods

Public methods:

- NCP$new()
- NCP$getStateful()
- NCP$setParams()
- NCP$getSites()
- NCP$setSites()
- NCP$addSite()
- NCP$cleanupInstance()
- NCP$run()
- NCP$clone()

Method new(): Create a new NCP object.

Usage:
NCP$new(ncp_defn, comp_defn, sites = list(), pubkey_bits = NULL, pubkey_n = NULL, den_bits = NULL)

Arguments:
- ncp_defn: the NCP definition; see example
- comp_defn: the computation definition
- sites: list of sites
- pubkey_bits: the number of bits in public key
- pubkey_n: the n for the public key
- den_bits: the number of bits in the denominator (power of 2) used in rational approximations

Returns: a new NCP object

Method getStateful(): Retrieve the value of the stateful field

Usage:
NCP$getStateful()

Method setParams(): Set some parameters of the NCP object for homomorphic computations

Usage:
NCP$setParams(pubkey_bits, pubkey_n, den_bits)

Arguments:
- pubkey_bits: the number of bits in public key
- pubkey_n: the n for the public key
- den_bits: the number of bits in the denominator (power of 2) used in rational approximations
**Method** `getSites()`: Retrieve the value of the private `sites` field

*Usage:*

```r
NCP$getSites()
```

**Method** `setSites()`: Set the value of the private `sites` field

*Usage:*

```r
NCP$setSites(sites)
```

*Arguments:*
- `sites` the list of sites

**Method** `addSite()`: Add a url or worker object for a site for participating in the distributed computation. The worker object can be used to avoid complications in debugging remote calls during prototyping.

*Usage:*

```r
NCP$addSite(name, url = NULL, worker = NULL)
```

*Arguments:*
- `name` of the site
- `url` web url of the site; exactly one of `url` or `worker` should be specified
- `worker` worker object for the site; exactly one of `url` or `worker` should be specified

**Method** `cleanupInstance()`: Clean up by destroying instance objects created in workspace.

*Usage:*

```r
NCP$cleanupInstance(token)
```

*Arguments:*
- `token` the token for the instance

**Method** `run()`: Run the distributed homomorphic computation

*Usage:*

```r
NCP$run(token)
```

*Arguments:*
- `token` a unique token for the run, used to ensure that correct parts of cached results are returned appropriately

*Returns:* the result of the computation

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

*Usage:*

```r
NCP$clone(deep = FALSE)
```

*Arguments:*
- `deep` Whether to make a deep clone.
QueryCountMaster

Create a master object to control worker objects generated by QueryCountWorker()

Description

QueryCountMaster objects instantiate and run a distributed query count computation

Methods

Public methods:

- QueryCountMaster$new()
- QueryCountMaster$kosher()
- QueryCountMaster$queryCount()
- QueryCountMaster$getSites()
- QueryCountMaster$addSite()
- QueryCountMaster$run()
- QueryCountMaster$clone()

Method new(): Create a new QueryCountMaster object.

Usage:
QueryCountMaster$new(defn, debug = FALSE)

Arguments:
defn the computation definition
debbug a flag for debugging, default FALSE

Returns: a new QueryCountMaster object

Method kosher(): Check if inputs and state of object are sane. For future use

Usage:
QueryCountMaster$kosher()

Returns: TRUE or FALSE

Method queryCount(): Run the distributed query count and return the result

Usage:
QueryCountMaster$queryCount()

Returns: the count

Method getSites(): Retrieve the value of the private sites field

Usage:
QueryCountMaster$getSites()

Method addSite(): Add a url or worker object for a site for participating in the distributed computation. The worker object can be used to avoid complications in debugging remote calls during prototyping.
Usage:
QueryCountMaster$addSite(name, url = NULL, worker = NULL)

Arguments:
name of the site
url web url of the site; exactly one of url or worker should be specified
worker worker object for the site; exactly one of url or worker should be specified

Method run(): Run the distributed query count

Usage:
QueryCountMaster$run()

Returns: the count

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

Usage:
QueryCountMaster$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also
QueryCountWorker() which goes hand-in-hand with this object

---

QueryCountWorker

*R6 worker object for use as a worker with master objects generated by
QueryCountMaster()*

---

Description

QueryCountWorker objects are worker objects at each site of a distributed QueryCount model computation

Methods

Public methods:

- QueryCountWorker$new()
- QueryCountWorker$getStateful()
- QueryCountWorker$kosher()
- QueryCountWorker$queryCount()
- QueryCountWorker$clone()

Method new(): Create a new QueryCountWorker object.

Usage:
QueryCountWorker$new(defn, data, stateful = FALSE)
**resetComputationInfo**

Clear the contents of the global store

**Arguments:**
- `defn` the computation definition
- `data` the local data
- `stateful` the statefulness flag, default FALSE

**Returns:** a new QueryCountWorker object

**Method getStateful():** Retrieve the value of the stateful field

**Usage:**
QueryCountWorker$getStateful()

**Method kosher():** Check if inputs and state of object are sane. For future use

**Usage:**
QueryCountWorker$kosher()

**Returns:** TRUE or FALSE

**Method queryCount():** Return the query count on the local data

**Usage:**
QueryCountWorker$queryCount()

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

**Usage:**
QueryCountWorker$clone(deep = FALSE)

**Arguments:**
- `deep` Whether to make a deep clone.

**See Also**
- QueryCountMaster() which goes hand-in-hand with this object

---

**Description**

In distcomp, several web applications need to communicate between themselves. Since only one application is expected to be active at any time, they do so via a global store, essentially a hash table. This function clears the store, except for the working directory.

**Usage**

resetComputationInfo()

**Value**

an empty list
runDistcompApp

See Also

setComputationInfo(), getComputationInfo()

runDistcompApp       Run a specified distcomp web application

Description

Web applications can define computation, setup worker sites or masters. This function invokes the appropriate web application depending on the task.

Usage

runDistcompApp(appType = c("definition", "setupWorker", "setupMaster"))

Arguments

appType       one of three values: "definition", "setupWorker", "setupMaster"

Value

the results of running the web application

See Also

defineNewComputation(), setupWorker(), setupMaster()

saveNewComputation       Save a computation instance, given the computation definition, associated data and possibly a data file name to use

Description

The function saveNewComputation uses the computation definition to save a new computation instance. This is typically done for every site that wants to participate in a computation with its own local data. The function examines the computation definition and uses the identifier therein to uniquely refer to the computation instance at the site. This function is invoked (maybe remotely) on the opencpu server by uploadNewComputation() when a worker site is being set up.

Usage

saveNewComputation(defn, data, dataFileName = NULL)
Arguments

defn an already defined computation
data the (local) data to use
dataFileName a file name to use for saving the data. Typically NULL, this is only needed when one is using a single opencpu server to behave like multiple sites in which case the data file name serves to distinguish the site-specific data files. When it is NULL, the data file name is taken from the configuration settings.

Value

TRUE if everything goes well

See Also

uploadNewComputation()
**setComputationInfo**

**See Also**

uploadNewNCP()

---

**setComputationInfo** *Set a name to a value in a global variable*

**Description**

In distcomp, several web applications need to communicate between themselves. Since only one application is expected to be active at any time, they do so via a global store, essentially a hash table. This function sets a name to a value.

**Usage**

`setComputationInfo(name, value)`

**Arguments**

- `name` the name for the object
- `value` the value for the object

**Value**

invisibly returns the all the name value pairs

**See Also**

getComputationInfo()

---

**setupMaster** *Setup a computation master*

**Description**

This function just calls `runDistcompApp()` with the parameter "setupMaster"

**Usage**

`setupMaster()`

**Value**

the results of running the web application

**See Also**

`runDistcompApp()`
**Description**

This function just calls `runDistcompApp()` with the parameter "setupWorker"

**Usage**

```r
setupWorker()
```

**Value**

the results of running the web application

**See Also**

`runDistcompApp()`

---

**SVDMaster**

*R6 class for SVD master object to control worker objects generated by SVDWorker()*

**Description**

SVDMaster objects instantiate and run a distributed SVD computation

**Methods**

**Public methods:**

- `SVDMaster$new()`
- `SVDMaster$kosher()`
- `SVDMaster$updateV()`
- `SVDMaster$updateU()`
- `SVDMaster$fixFit()`
- `SVDMaster$reset()`
- `SVDMaster$addSite()`
- `SVDMaster$run()`
- `SVDMaster$summary()`
- `SVDMaster$clone()`

**Method** `new()`: SVDMaster objects instantiate and run a distributed SVD computation

**Usage:**

```r
SVDMaster$new(defn, debug = FALSE)
```
Method kosher(): Check if inputs and state of object are sane. For future use
Usage:
SVDMaster$kosher()
Returns: TRUE or FALSE

Method updateV(): Return an updated value for the V vector, normalized by arg
Usage:
SVDMaster$updateV(arg)
Arguments:
arg the normalizing value
... other args ignored
Returns: updated V

Method updateU(): Update U and return the updated norm of U
Usage:
SVDMaster$updateU(arg)
Arguments:
arg the normalizing value
... other args ignored
Returns: updated norm of U

Method fixFit(): Construct the residual matrix using given the V vector and d so far
Usage:
SVDMaster$fixFit(v, d)
Arguments:
v the value for v
d the value for d
Returns: result

Method reset(): Reset the computation state by initializing work matrix and set up starting values for iterating
Usage:
SVDMaster$reset()

Method addSite(): Add a url or worker object for a site for participating in the distributed computation. The worker object can be used to avoid complications in debugging remote calls during prototyping.
Usage:
SVDMaster$addSite(name, url = NULL, worker = NULL)

Arguments:
name  of the site
url  web url of the site; exactly one of url or worker should be specified
worker  worker object for the site; exactly one of url or worker should be specified

Method run(): Run the distributed Cox model fit and return the estimates

Usage:
SVDMaster$run(thr = 1e-08, max.iter = 100)

Arguments:
thr  the threshold for convergence, default 1e-8
max.iter  the maximum number of iterations, default 100

Returns: a named list of \( V, d \)

Method summary(): Return the summary result

Usage:
SVDMaster$summary()

Returns: a named list of \( V, d \)

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

Usage:
SVDMaster$clone(deep = FALSE)

Arguments:
deep  Whether to make a deep clone.

See Also

SVDWorker() which goes hand-in-hand with this object

---

**SVDWorker**  
*R6 class for a SVD worker object to use with master objects generated by SVDMaster()*

**Description**

SVDWorker objects are worker objects at each site of a distributed SVD model computation
Methods

Public methods:

• `SVDWorker$new()`  
• `SVDWorker$reset()`  
• `SVDWorker$dimX()`  
• `SVDWorker$updateV()`  
• `SVDWorker$updateU()`  
• `SVDWorker$normU()`  
• `SVDWorker$fixU()`  
• `SVDWorker$getN()`  
• `SVDWorker$getP()`  
• `SVDWorker$getStateful()`  
• `SVDWorker$kosher()`  
• `SVDWorker$clone()`

Method `new()`: Create a new SVDWorker object.

Usage:

`SVDWorker$new(defn, data, stateful = TRUE)`

Arguments:

`defn` the computation definition  
`data` the local x matrix  
`stateful` a boolean flag indicating if state needs to be preserved between REST calls, `TRUE` by default

Returns: a new SVDWorker object

Method `reset()`: Reset the computation state by initializing work matrix and set up starting values for iterating

Usage:

`SVDWorker$reset()`

Method `dimX()`: Return the dimensions of the matrix

Usage:

`SVDWorker$dimX(...)`

Arguments:

... other args ignored

Returns: the dimension of the matrix

Method `updateV()`: Return an updated value for the V vector, normalized by arg

Usage:

`SVDWorker$updateV(arg, ...)`

Arguments:

`arg` the normalizing value
... other args ignored

Returns: updated \( V \)

**Method** `updateU()`: Update \( U \) and return the updated norm of \( U \)

Usage:
`SVDWorker$updateU(arg, ...)`

Arguments:
arg the initial value
... other args ignored

Returns: updated norm of \( U \)

**Method** `normU()`: Normalize \( U \) vector

Usage:
`SVDWorker$normU(arg, ...)`

Arguments:
arg the normalizing value
... other args ignored

Returns: TRUE invisibly

**Method** `fixU()`: Construct residual matrix using \( arg \)

Usage:
`SVDWorker$fixU(arg, ...)`

Arguments:
arg the value to use for residualizing
... other args ignored

**Method** `getN()`: Get the number of rows of \( x \) matrix

Usage:
`SVDWorker$getN()`

Returns: the number of rows of \( x \) matrix

**Method** `getP()`: Get the number of columns of \( x \) matrix

Usage:
`SVDWorker$getP()`

Returns: the number of columns of \( x \) matrix

**Method** `getStateful()`: Return the stateful status of the object.

Usage:
`SVDWorker$getStateful()`

Returns: the stateful flag, TRUE or FALSE

**Method** `kosher()`: Check if inputs and state of object are sane. For future use
Usage:
SVDWorker$kosher()

Returns: TRUE or FALSE

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

Usage:
SVDWorker$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

See Also
SVDMaster() which goes hand-in-hand with this object

uploadNewComputation  Upload a new computation and data to an opencpu server

Description
The function uploadNewComputation is really a remote version of saveNewComputation(), invoking that function on an opencpu server. This is typically done for every site that wants to participate in a computation with its own local data. Note that a site is always a list of at least a unique name element (distinguishing the site from others) and a url element.

Usage
uploadNewComputation(site, defn, data)

Arguments

site a list of two items, a unique name and a url
defn the identifier of an already defined computation
data the (local) data to use

Value
TRUE if everything goes well

See Also
saveNewComputation()
uploadNewNCP  
*Upload a new Non-Cooperating Party (NCP) information and sites to an opencpu server*

**Description**

The function `uploadNewNCP` is really a remote version of `saveNewNCP()`, invoking that function on an opencpu server. This is typically done for the two NCPs participating in a computation with the list of sites. Note that sites are always a list of at least a unique name element (distinguishing the site from others) and a url element.

**Usage**

```r
uploadNewNCP(defn, comp_defn, url = NULL, worker = NULL, sites)
```

**Arguments**

- `defn`: a definition for the NCP
- `comp_defn`: the computation definition
- `url`: the url for the NCP. Only one of url and worker can be non-null
- `worker`: the worker for the NCP if local. Only one of url and worker can be non-null
- `sites`: a list of lists, each containing two items, a unique name and a (not necessarily unique) url. This is the data for the NCP!

**Value**

TRUE if everything goes well

**See Also**

- `saveNewNCP()`

---

writeCode  
*Write the code necessary to run a master process*

**Description**

Once a computation is defined, worker sites are set up, the master process code is written by this function. The current implementation does not allow one to mix localhost URLs with non-localhost URLs.

**Usage**

```r
writeCode(defn, sites, outputFilenamePrefix)
```
writeCode

Arguments

- **defn**: the computation definition
- **sites**: a named list of site URLs participating in the computation
- **outputFilenamePrefix**: the name of the output file prefix using which code and data will be written

Value

the value TRUE if all goes well

See Also

- `setupMaster()`
availableComputations, 3
availableComputations(), 8, 10, 22, 23, 25
availableDataSources, 4
CoxMaster, 4
CoxWorker, 6
createHEWorkerInstance, 7
createNCPInstance, 8
createWorkerInstance, 9
createWorkerInstance(), 11
defineNewComputation, 10
defineNewComputation(), 31
destroyInstanceObject, 11
digest::digest(), 15
distcomp, 11
distcompSetup, 12
distcompSetup(), 12, 16
executeHEMethod, 14
executeMethod, 14
generateId, 15
getComputationInfo, 15
getComputationInfo(), 3, 31, 33
getConfig, 16
getConfig(), 13
HEMaster, 17
HEQueryCountMaster, 18
HEQueryCountMaster(), 20, 22
HEQueryCountWorker, 20
HEQueryCountWorker(), 18, 20
makeDefinition, 22
makeHEMaster, 23
makeMaster, 23
makeNCP, 24
makeWorker, 24
NCP, 25
QCP(), 18
QueryCountMaster, 28
QueryCountMaster(), 29, 30
QueryCountWorker, 29
QueryCountWorker(), 28, 29
resetComputationInfo, 30
runDistcompApp, 31
runDistcompApp(), 10, 33, 34
saveNewComputation, 31
saveNewComputation(), 39
saveNewNCP, 32
saveNewNCP(), 40
setComputationInfo, 33
setComputationInfo(), 15, 31
setupMaster, 33
setupMaster(), 31, 41
setupWorker, 34
setupWorker(), 31
survival::coxph(), 12
SVDMaster, 34
SVDMaster(), 36, 39
SVDWorker, 36
SVDWorker(), 34, 36
uploadNewComputation, 39
uploadNewComputation(), 31, 32
uploadNewNCP, 40
uploadNewNCP(), 33
writeCode, 40