Package ‘distcomp’

May 16, 2017

Title Computations over Distributed Data without Aggregation

Maintainer Balasubramanian Narasimhan <naras@stat.Stanford.EDU>

Version 1.0-1

URL http://dx.doi.org/10.18637/jss.v077.i13

Depends survival, stats, R (>= 3.1.0)

Imports utils, shiny, httr (>= 1.0.0), digest, jsonlite, stringr, R6 (>= 2.0)

Suggests opencpu

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. Web applications are provided (via 'shiny') for the implemented methods to help in designing and deploying the computations.

Copyright inst/COPYRIGHTS

License LGPL (>= 2)

RoxygenNote 6.0.1

NeedsCompilation yes

Author Balasubramanian Narasimhan [aut, cre],
Marina Bendersky [aut],
Sam Gross [aut],
Terry M. Therneau [ctb],
Thomas Lumley [ctb]

Repository CRAN

Date/Publication 2017-05-16 19:08:02 UTC

R topics documented:

`availableComputations` *Return the currently available (implemented) computations*

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 worker objects generated by* CoxWorker

---

Description

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

Usage

```
CoxMaster
```

Format

An `R6Class` generator object

Methods

`CoxMaster$new(defnId, formula, debug=FALSE)` Create a new CoxMaster object using the defnId and formula. The debug flag is useful for debugging

`logLik(beta, ...)` Compute the partial log likelihood for all the data by aggregating the values at each site. The return value is numeric scalar with two attributes: `gradient` contains the score vector, and `hessian` contains the estimated hessian matrix

`addSite(name, url)` Add a worker site for participating in the distributed computation

`var(beta, ...)` Compute the variance of the parameter vector beta

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

`getP()` Returns the dimension of the parameter vector

`run()` Run the fitting iterations and save the result

`summary()` Return 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

See Also

`CoxWorker` which generates objects matched to such a master object

CoxWorker — *Create a worker object for use as a worker with master objects generated by CoxMaster*

Description

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

Usage

```
CoxWorker
```

Format

An `R6Class` generator object

Methods

`CoxWorker$new(formula, data, stateful=TRUE)` Create a new CoxWorker instance object using formuala and data. The stateful flag indicates whether the object state is to be saved between iterations

`logLik(beta, ...)` Compute the partial log likelihood for the local data for the input parameter vector beta. The return value is 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

`var(beta, ...)` Compute the variance of the parameter vector beta

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

`getP()` Returns the dimension of the parameter vector

`getStateful()` Returns `TRUE` if object is stateful, else `FALSE`

See Also

`CoxMaster` which goes hand-in-hand with this object

createInstanceObject — *Given the definition identifier of an object, instantiate and store object in workspace*

Description

The function `createInstanceObject` 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

Usage

```
createInstanceObject(defnId, instanceId, dataFileName = NULL)
```

Arguments

`defnId`	the identifier of an already defined computation
`instanceId`	an indentifier to use for the created instance
`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`

`defineNewComputation` *Define a new computation*

Description

This function just calls `runDistcompApp` with the parameter "definition"

Usage

```
defineNewComputation()
```

Value

the results of running the web application

See Also

`runDistcompApp`

`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

`createInstanceObject`

`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 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

Software for Distributed Computation on Medical Databases: A Demonstration Project. Journal of Statistical Software, 77(13), 1-22. doi:10.18637/jss.v077.i13

Appendix E of Modeling Survival Data: Extending the Cox Model by Terry M. Therneau and Patricia Grambsch. Springer Verlag, 2000.

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 discompsetup 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

```
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",
  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
`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)
```

`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

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

Arguments

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

Value

the hash as a string

See Also

`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

```
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

```
## Not run:
getConfig()

## End(Not run)
```

`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

```
## Not run:
makeDefinition(names(availableComputations())[1])

## End(Not run)
```

---

makeMaster *Make a master object given a definition*

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)
```

Arguments

defn	the computation definition

Value

a master object of the appropriate class based on the definition

See Also

availableComputations

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

```
makeWorker(defn, data)
```

Arguments

defn	the computation definition
data	the data for the computation

Value

a worker object of the appropriate class based on the definition

See Also

availableComputations

resetComputationInfo *Clear the contents of 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 clears the store, except for the working directory.

Usage

```
resetComputationInfo()
```

Value

an empty list

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`	the identifier of 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 *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

---

setupWorker — *Setup a worker site*

Description

This function just calls runDistcompApp with the parameter "setupWorker"

Usage

```
setupWorker()
```

Value

the results of running the web application

See Also

runDistcompApp

SVDMaster — *Create a master object to control worker objects generated by SVDWorker*

Description

SVDMaster objects instantiate and run a distributed SVD computation

Usage

```
SVDMaster
```

Format

An R6Class generator object

Methods

`SVDMaster$new(defnId, k, debug=FALSE)` Create an SVD master object with the specified id, the number of singular vectors desired, and the debugging flag. The debug flag is used for debugging computations

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

`updateV(arg)` Return an updated value for the V vector

`updateU(arg)` Return an updated value for the U vector

`fixFit(v, d)` Construct the residual matrix using given the v vector and d so far

`reset()` Initialize the computation

`dimX(, ...)` Return the dimensions of the matrix

`normU(arg, ...)` Normalize U vector by `arg`

`addSite(name, url)` Add a worker site for participating in the distributed computation

`run(k = private$k, thr = 1e-8, max.iter = 100)` Run the SVD computation until either the threshold is reached or the max.iter number of iterations are used up

`summary()` Return the summary of results

See Also

`SVDWorker` which goes hand-in-hand with this object

`SVDWorker` *Create a worker object for use as a worker with master objects generated by* `SVDMaster`

Description

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

Usage

```
SVDWorker
```

Format

An `R6Class` generator object

Methods

`SVDWorker$new(x, stateful=TRUE)` Create a new SVD worker object with data x and flag for preserving state between iterations

`reset()` Initialize work matrix and set up starting values for iterating

`dimX(...)` Return the dimensions of the matrix

`updateV(arg, ...)` Return an updated value for the V vector, normalized by `arg` `updateU(arg, ...)` Return an updated value for the norm of the U vector

`normU(arg, ...)` Normalize U vector by `arg`

`fixU(arg, ...)` Construct the residual matrix using `arg`

`getN(...)` Return the number of rows

`getP(...)` Return the number of columns

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

`getStateful()` Returns `TRUE` if object is stateful, else `FALSE`

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`

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

```
writeCode(defn, sites, outputFileName)
```

Arguments

defn	the computation definition
sites	a named list of site URLs participating in the computation
outputFileName	the name of the output file to which code will be written

Value

the value `TRUE` if all goes well

See Also

setupMaster

Index