Package 'dsBaseClient'

January 7, 2026

Title 'DataSHIELD' Client Side Base Functions

Version 6.3.5

Description Base 'DataSHIELD' functions for the client side. 'DataSHIELD' is a software package which allows
you to do non-
disclosive federated analysis on sensitive data. 'DataSHIELD' analytic functions have been designed to only share non disclosive summary statistics, with built in automated output checking based on statistical disclosure control. With data sites setting the threshold values for the automated output checks. For more details, see citation('dsBaseClient').

License GPL-3

Depends R (>= 4.0.0), DSI (>= 1.7.1)

Imports fields, metafor, meta, ggplot2, gridExtra, data.table, methods, dplyr

Suggests lme4, httr, spelling, tibble, testthat, e1071, DescTools, DSOpal, DSMolgenisArmadillo, DSLite

RoxygenNote 7.3.3

Encoding UTF-8

Language en-GB

NeedsCompilation no

Author Paul Burton [aut] (ORCID: <https://orcid.org/0000-0001-5799-9634>),
Rebecca Wilson [aut] (ORCID: <https://orcid.org/0000-0003-2294-593X>),
Olly Butters [aut] (ORCID: <https://orcid.org/0000-0003-0354-8461>),
Patricia Ryser-Welch [aut] (ORCID:
<https://orcid.org/0000-0002-0070-0264>),
Alex Westerberg [aut],
Leire Abarrategui [aut],
Roberto Villegas-Diaz [aut] (ORCID:
<https://orcid.org/0000-0001-5036-8661>),
Demetris Avraam [aut] (ORCID: <https://orcid.org/0000-0001-8908-2441>),
Yannick Marcon [aut] (ORCID: <https://orcid.org/0000-0003-0138-2023>),
Tom Bishop [aut],
Amadou Gaye [aut] (ORCID: <https://orcid.org/0000-0002-1180-2792>),
Xavier Escribà-Montagut [aut] (ORCID:

<https://orcid.org/0000-0003-2888-8948>),
Stuart Wheater [aut, cre] (ORCID:
<https://orcid.org/0009-0003-2419-1964>)

Maintainer Stuart Wheater <stuart.wheater@arjuna.com>

Repository CRAN

Date/Publication 2026-01-07 08:20:07 UTC

Contents

`ds.abs` *Computes the absolute values of a variable*

Description

Computes the absolute values for a specified numeric or integer vector. This function is similar to R function `abs`.

Usage

```
ds.abs(x = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x	a character string providing the name of a numeric or an integer vector.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default name is set to abs.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The function calls the server-side function absDS that computes the absolute values of the elements of a numeric or integer vector and assigns a new vector with those absolute values on the server-side. The name of the new generated vector is specified by the user through the argument newobj, otherwise is named by default to abs.newobj.

Value

ds.abs assigns a vector for each study that includes the absolute values of the input numeric or integer vector specified in the argument x. The created vectors are stored in the servers.

Author(s)

Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
```

```
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Generate a normally distributed variable with zero mean and variance equal
  #  to one and then get their absolute values
  ds.rNorm(samp.size=100, mean=0, sd=1, newobj='var.norm', datasources=connections)
  # check the quantiles
  ds.summary(x='var.norm', datasources=connections)
  ds.abs(x='var.norm', newobj='var.norm.abs', datasources=connections)
  # check now the changes in the quantiles
  ds.summary(x='var.norm.abs', datasources=connections)

  # Example 2: Generate a sequence of negative integer numbers from -200 to -100
  # and then get their absolute values
  ds.seq(FROM.value.char = '-200', TO.value.char = '-100', BY.value.char = '1',
         newobj='negative.integers', datasources=connections)
  # check the quantiles
  ds.summary(x='negative.integers', datasources=connections)
  ds.abs(x='negative.integers', newobj='positive.integers', datasources=connections)
  # check now the changes in the quantiles
  ds.summary(x='positive.integers', datasources=connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.asCharacter` *Converts a server-side R object into a character class*

Description

Converts the input object into a character class. This function is based on the native R function `as.character`.

Usage

```
ds.asCharacter(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x.name`	a character string providing the name of the input object to be coerced to class character.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `ascharacter.newobj`.

datasources — a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Server function called: asCharacterDS

Value

ds.asCharacter returns the object converted into a class character that is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the newobj which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Converting the R object into a class character
  ds.asCharacter(x.name = "D$LAB_TSC",
                 newobj = "char.obj",
              datasources = connections[1]) #only the first Opal server is used ("study1")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)
```

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

---

ds.asDataMatrix *Converts a server-side R object into a matrix*

Description

Coerces an R object into a matrix maintaining original class for all columns in data frames.

Usage

```
ds.asDataMatrix(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x.name	a character string providing the name of the input object to be coerced to a matrix.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default asdatamatrix.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is based on the native R function data.matrix.

Server function called: asDataMatrixDS.

Value

ds.asDataMatrix returns the object converted into a matrix that is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the newobj which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
```

```
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Converting the R object into a matrix
  ds.asDataMatrix(x.name = "D",
                  newobj = "mat.obj",
              datasources = connections[1]) #only the first Opal server is used ("study1")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.asFactor` *Converts a server-side numeric vector into a factor*

Description

This function assigns a server-side numeric vector into a factor class.

Usage

```
ds.asFactor(
  input.var.name = NULL,
  newobj.name = NULL,
  forced.factor.levels = NULL,
  fixed.dummy.vars = FALSE,
  baseline.level = 1,
  datasources = NULL
)
```

Arguments

input.var.name	a character string which provides the name of the variable to be converted to a factor.
newobj.name	a character string that provides the name for the output variable that is stored on the data servers. Default asfactor.newobj.
forced.factor.levels	the levels that the user wants to split the input variable. If NULL (default) a vector with all unique levels from all studies are created.
fixed.dummy.vars	boolean. If TRUE the input variable is converted to a factor but presented as a matrix of dummy variables. If FALSE (default) the input variable is converted to a factor and assigned as a vector.
baseline.level	an integer indicating the baseline level to be used in the creation of the matrix with dummy variables. If the fixed.dummy.vars is set to FALSE then any value of the baseline level is not taken into account.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Converts a numeric vector into a factor type which is represented either as a vector or as a matrix of dummy variables depending on the argument fixed.dummy.vars. The matrix of dummy variables also depends on the argument baseline.level.

ds.asFactor.R and its associated serverside functions asFactorDS1 and asFactorDS2 are to be used when you have variable that has up to 40 unique levels across all sources combined. If one of the sources does not contain any subjects at a particular level, that level will still be created as an empty category. In the end all sources thus include a factor variable with consistent factor levels across all sources - one level for every unique value that occurs in at least one source. This is important when you wish to fit models using ds.glm because the factor levels must be consistent across all studies or the model will not fit.

But in order for this to be possible, all sources have to share all of the unique values their source holds for the variable. This allows the client to create a single vector containing all of the unique factor levels across ALL sources. But this is potentially disclosive if there are too many levels. There are therefore two checks on the number of levels in each source. One is simply a test of whether the number of levels exceeds a value specified by the Roption value 'nfilter.max.levels' which is set by default to 40, but the data custodian for the source can choose any alternative value he/she chooses. The second test is of whether the levels are too dense: ie do the number of levels exceed a specified proportion of the full length of the relevant vector in the particular source. The max density is set by the Roption value 'nfilter.levels' which takes the default value 0.33 but can again be modified by the data custodian.

In combination, these two checks mean that if a factor has 35 levels in a given study where the total length of the variable to be converted to a factor is 1000 individuals, the ds.asFactor function will process that variable appropriately. But if it had had 45 levels it would have been blocked by 'nfilter.max.levels' and if the total length of the variable in that study had only been 70 subjects it would have been blocked by the density criterion held in 'nfilter.levels'.

If you have a factor with more than 40 levels in each source - perhaps most commonly an ID of some sort that you need to provide as an argument to eg a tapply function. Then you cannot use ds.asFactor. Typically in these circumstance you simply want to create a factor that is appropriate for each source but you do not need to ensure that all levels are consistent across all sources. In that case, you can use the ds.asFactorSimple function which does no more than coerce a numeric or character variable to a factor. Because you do not need to share unique factor levels between sources, there is then no disclosure issue.

To understand how the matrix of the dummy variable is created let's assume that we have the vector `(1, 2, 1, 3, 4, 4, 1, 3, 4, 5)` of ten integer numbers. If we set the argument `fixed.dummy.vars = TRUE`, `baseline.level = 1` and `forced.factor.levels = c(1,2,3,4,5)`. The input vector is converted to the following matrix of dummy variables:

DV2	DV3	DV4	DV5
0	0	0	0
1	0	0	0
0	0	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

For the same example if the `baseline.level = 3` then the matrix is:

DV1	DV2	DV4	DV5
1	0	0	0
0	1	0	0
1	0	0	0
0	0	0	0
0	0	1	0
0	0	1	0
1	0	0	0
0	0	0	0
0	0	1	0
0	0	0	1

In the first instance the first row of the matrix has zeros in all entries indicating that the first data point belongs to level 1 (as the baseline level is equal to 1). The second row has 1 at the first (`DV2`) column and zeros elsewhere, indicating that the second data point belongs to level 2. In the second instance (second matrix) where the baseline level is equal to 3, the first row of the matrix has 1 at the first (`DV1`) column and zeros elsewhere, indicating again that the first data point belongs to level 1. Also as we can see the fourth row of the second matrix has all its elements equal to zero indicating that the fourth data point belongs to level 3 (as the baseline level, in that case, is 3).

If the `baseline.level` is set to be equal to a value that is not one of the levels of the factor then a matrix of dummy variables is created having as many columns as the number of levels. In that case in each row there is a unique entry equal to 1 at a certain column indicating the level of each data

point. So, for the above example where the vector has five levels if we set the `baseline.level` equal to a value that does not belong to those five levels (`baseline.level=8`) the matrix of dummy variables is:

DV1	DV2	DV3	DV4	DV5
1	0	0	0	0
0	1	0	0	0
1	0	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	1	0
1	0	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Server functions called: `asFactorDS1` and `asFactorDS2`

Value

`ds.asFactor` returns the unique levels of the converted variable in ascending order and a validity message with the name of the created object on the client-side and the output matrix or vector in the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  ds.asFactor(input.var.name = "D$PM_BMI_CATEGORICAL",
              newobj.name = "fact.obj",
              forced.factor.levels = NULL, #a vector with all unique levels
                                           #from all studies is created
              fixed.dummy.vars = TRUE, #create a matrix of dummy variables
              baseline.level = 1,
              datasources = connections)#all the Opal servers are used, in this case 3
                                        #(see above the connection to the servers)
  ds.asFactor(input.var.name = "D$PM_BMI_CATEGORICAL",
              newobj.name = "fact.obj",
              forced.factor.levels = c(2,3), #the variable is split in 2 levels
              fixed.dummy.vars = TRUE, #create a matrix of dummy variables
              baseline.level = 1,
             datasources = connections[1])#only the first Opal server is used ("study1")

   # Clear the Datashield R sessions and logout
   datashield.logout(connections)

## End(Not run)
```

`ds.asFactorSimple` *Converts a numeric vector into a factor*

Description

ds.asFactorSimple calls the assign function asFactorSimpleDS and thereby coerces a numeric or character vector into a factor

Usage

```
ds.asFactorSimple(
  input.var.name = NULL,
  newobj.name = NULL,
  datasources = NULL
)
```

Arguments

`input.var.name`	a character string which provides the name of the variable to be converted to a factor.
`newobj.name`	a character string that provides the name for the output variable that is stored on the data servers. Default `asfactor.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The function converts the input variable into a factor. Unlike ds.asFactor and its serverside functions, ds.asFactorSimple does no more than coerce the class of a variable to make it a factor on the serverside in each data source. It does not check for or enforce consistency of factor levels across sources or allow you to force an arbitrary set of levels unless those levels actually exist in the sources. Furthermore, it does not allow you to create an array of binary dummy variables that is equivalent to a factor. If you need to do any of these things you will have to use the ds.asFactor function.

Value

an output vector of class factor to the serverside. In addition, returns a validity message with the name of the created object on the client-side and if creation fails an error message which can be viewed using datashield.errors().

Author(s)

DataSHIELD Development Team

`ds.asInteger` *Converts a server-side R object into an integer class*

Description

Coerces an R object into an integer class. This function is based on the native R function `as.integer`.

Usage

```
ds.asInteger(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x.name`	a character string providing the name of the input object to be coerced to an integer.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `asinteger.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function is based on the native R function `as.integer`. The only difference is that the DataSHIELD function first converts the values of the input object into characters and then convert those to integers. This addition, it is important for the case where the input object is of class factor having integers as levels. In that case, the native R `as.integer` function returns the underlying level codes and not the values as integers. For example `as.integer` in R converts the factor vector:

[1] 0 1 1 2 1 0 1 0 2 2 2 1
Levels: 0 1 2
to the following integer vector: 1 2 2 3 2 1 2 1 3 3 3 2

Server function called: `asIntegerDS`

Value

`ds.asInteger` returns the R object converted into an integer that is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the `newobj` which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Converting the R object into an integer
  ds.asInteger(x.name = "D$LAB_TSC",
                  newobj = "int.obj",
              datasources = connections[1]) #only the first Opal server is used ("study1")
  ds.class(x = "int.obj", datasources = connections[1])

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)
```

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

---

ds.asList *Converts a server-side R object into a list*

Description

Coerces an R object into a list. This function is based on the native R function `as.list`.

Usage

```
ds.asList(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x.name`	a character string providing the name of the input object to be coerced to a list.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `aslist.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Server function called: `asListDS`

Value

`ds.asList` returns the R object converted into a list which is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the `newobj` which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
builder <- DSI::newDSLoginBuilder()
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Converting the R object into a List
ds.asList(x.name = "D",
newobj = "D.asList",
datasources = connections[1]) #only the first Opal server is used ("study1")
ds.class(x = "D.asList", datasources = connections[1])

# Clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.asLogical` *Converts a server-side R object into a logical class*

Description

Coerces an R object into a logical class. This function is based on the native R function `as.logical`.

Usage

```
ds.asLogical(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x.name`	a character string providing the name of the input object to be coerced to a logical.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `aslogical.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Server function called: `asLogicalDS`

Value

`ds.asLogical` returns the R object converted into a logical that is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the `newobj` which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Converting the R object into a logical
  ds.asLogical(x.name = "D$LAB_TSC",
               newobj = "logical.obj",
              datasources =connections[1]) #only the first Opal server is used ("study1")
  ds.class(x = "logical.obj", datasources = connections[1])

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

ds.asMatrix *Converts a server-side R object into a matrix*

Description

Coerces an R object into a matrix. This converts all columns into character class.

Usage

```
ds.asMatrix(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x.name	a character string providing the name of the input object to be coerced to a matrix.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default asmatrix.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is based on the native R function as.matrix. If this function is applied to a data frame, all columns are converted into a character class. If you wish to convert a data frame to a matrix but maintain all data columns in their original class you should use the function ds.asDataMatrix.

Server function called: asMatrixDS

Value

ds.asMatrix returns the object converted into a matrix that is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the newobj which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
builder <- DSI::newDSLoginBuilder()
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Converting the R object into a matrix
ds.asMatrix(x.name = "D",
            newobj = "mat.obj",
           datasources = connections[1]) #only the first Opal server is used ("study1")

# Clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.asNumeric` *Converts a server-side R object into a numeric class*

Description

Coerces an R object into a numeric class. This function is based on the native R function `as.numeric`.

Usage

```
ds.asNumeric(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x.name`	a character string providing the name of the input object to be coerced to a numeric.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `asnumeric.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function is based on the native R function `as.numeric`. However, it behaves differently with some specific classes of variables. For example, if the input object is of class factor, it first converts its values into characters and then convert those to numerics. This behaviour is important for the case where the input object is of class factor having numbers as levels. In that case, the native R `as.numeric` function returns the underlying level codes and not the values as numbers. For example `as.numeric` in R converts the factor vector:
0 1 1 2 1 0 1 0 2 2 2 1
Levels: 0 1 2
to the following numeric vector: 1 2 2 3 2 1 2 1 3 3 3 2
In contrast DataSHIELD converts an input factor with numeric levels to its original numeric values.

Server function called: `asNumericDS`

Value

`ds.asNumeric` returns the R object converted into a numeric class that is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the `newobj` which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")
```

```
# Converting the R object into a numeric class
ds.asNumeric(x.name = "D$LAB_TSC",
                newobj = "num.obj",
            datasources = connections[1]) #only the first Opal server is used ("study1")
ds.class(x = "num.obj", datasources = connections[1])

# Clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.assign` *Assigns an R object to a name in the server-side*

Description

This function assigns a datashield object to a name, hence creating a new object.

Usage

```
ds.assign(toAssign = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`toAssign`	a character string providing the object to assign.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `assign.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The new object is stored on the server-side.

`ds.assign` causes a remote assignment by using `DSI::datashield.assign`. The `toAssign` argument is checked at the server and assigned the variable called `newobj` on the server-side.

Value

`ds.assign` returns the R object assigned to a name that is written to the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Assign a variable to a name
  ds.assign(toAssign = "D$LAB_TSC",
            newobj = "labtsc",
            datasources = connections[1]) #only the first Opal server is used ("study1")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.auc *Calculates the Area under the curve (AUC)*

Description

This function calculates the C-statistic or AUC for logistic regression models.

Usage

```
ds.auc(pred = NULL, y = NULL, datasources = NULL)
```

Arguments

pred	the name of the vector of the predicted values
y	the name of the outcome variable. Note that this variable should include the complete cases that are used in the regression model.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The AUC determines the discriminative ability of a model.

Value

returns the AUC and its standard error

Author(s)

Demetris Avraam for DataSHIELD Development Team

---

ds.Boole *Converts a server-side R object into Boolean indicators*

Description

It compares R objects using the standard set of Boolean operators (==, !=, >, >=, <, <=) to create a vector with Boolean indicators that can be of class logical (TRUE/FALSE) or numeric (1/0).

Usage

```
ds.Boole(
  V1 = NULL,
  V2 = NULL,
  Boolean.operator = NULL,
  numeric.output = TRUE,
  na.assign = "NA",
  newobj = NULL,
  datasources = NULL
)
```

Arguments

V1	A character string specifying the name of the vector to which the Boolean operator is to be applied.
V2	A character string specifying the name of the vector to compare with V1.
Boolean.operator	A character string specifying one of six possible Boolean operators: '==', '!=', '>', '>=', '<' and '<='.

numeric.output	logical. If TRUE the output variable should be of class numeric (1/0). If FALSE the output variable should be of class logical (TRUE/FALSE). Default TRUE.
na.assign	A character string taking values 'NA','1' or '0'. Default 'NA'. For more information see details.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default boole.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

A combination of different Boolean operators using AND operator can be obtained by multiplying two or more binary/Boolean vectors together. In this way, observations taking the value 1 in every vector will then take the value 1 in the final vector (after multiplication) while all others will take the value 0. Instead the combination using OR operator can be obtained by the sum of two or more vectors and applying ds.Boole using the operator >= 1.

In na.assign if 'NA' is specified, the missing values remain as NAs in the output vector. If '1' or '0' is specified the missing values are converted to 1 or 0 respectively or TRUE or FALSE depending on the argument numeric.output.

Server function called: BooleDS

Value

ds.Boole returns the object specified by the newobj argument which is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the newobj which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
```

```
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Generating Boolean indicators
  ds.Boole(V1 = "D$LAB_TSC",
           V2 = "D$LAB_TRIG",
           Boolean.operator = ">",
           numeric.output = TRUE, #Output vector of 0 and 1
           na.assign = "NA",
           newobj = "Boole.vec",
           datasources = connections[1]) #only the first server is used ("study1")

  ds.Boole(V1 = "D$LAB_TSC",
           V2 = "D$LAB_TRIG",
           Boolean.operator = "<",
           numeric.output = FALSE, #Output vector of TRUE and FALSE
           na.assign = "1", #NA values are converted to TRUE
           newobj = "Boole.vec",
           datasources = connections[2]) #only the second server is used ("study2")

  ds.Boole(V1 = "D$LAB_TSC",
           V2 = "D$LAB_TRIG",
           Boolean.operator = ">",
           numeric.output = TRUE, #Output vector of 0 and 1
           na.assign = "0", #NA values are converted to 0
           newobj = "Boole.vec",
           datasources = connections) #All servers are used

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.boxPlot` *Draw boxplot*

Description

Draw boxplot with data on the study servers (data frames or numeric vectors) with the option of grouping using categorical variables on the dataset (only for data frames)

Usage

```
ds.boxPlot(
  x,
  variables = NULL,
  group = NULL,
  group2 = NULL,
  xlabel = "x axis",
  ylabel = "y axis",
  type = "pooled",
  datasources = NULL
)
```

Arguments

x	character Name of the data frame (or numeric vector) on the server side that holds the information to be plotted
variables	character vector Name of the column(s) of the data frame to include on the boxplot
group	character (default NULL) Name of the first grouping variable.
group2	character (default NULL) Name of the second grouping variable.
xlabel	caracter (default "x axis") Label to put on the x axis of the plot
ylabel	caracter (default "y axis") Label to put on the y axis of the plot
type	character Return a pooled plot ("pooled") or a split plot (one for each study server "split")
datasources	a list of DSConnection-class (default NULL) objects obtained after login

Value

ggplot object

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  ### Please ensure you have a training Virtual Machine running,
  # or that you have a live connection to a server.

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE,
 symbol = "D")

## Create a boxplot of one variable
ds.boxPlot("D", "LAB_HDL", datasources = connections)

## Create a boxplot that is split by study:
ds.boxPlot("D", "LAB_HDL", type= "split", datasources = connections)

## Create a boxplot of two variables variable
ds.boxPlot("D", c("LAB_HDL", "LAB_TRIG"), type="pooled",
datasources = connections)
# only one plot is created (of the aggregated results of all servers)

## Create a boxplot of two variables, which are split by a factor
ds.boxPlot("D", c("LAB_HDL", "LAB_TRIG"), group = "GENDER",
datasources = connections)

## Create a boxplot with x- and y-axis labels
ds.boxPlot("D", c("LAB_HDL", "LAB_TRIG"), group = "GENDER",
xlabel = "Variable", ylabel = "Measurement", datasources = connections)

## Improve the presentation of ds.boxplot output using ggplot:
### User must save the output, which is in a ggplot format already:
a <- ds.boxPlot("D", c("LAB_HDL", "LAB_TRIG"), group = "GENDER",
xlabel = "Variable", ylabel = "Measurement", datasources = connections)

### Then customise output "a" using ggplot tools:
a + ggplot2::scale_fill_discrete(name = "Gender", labels = c("Male", "Female"))

### Or use an alternative way, to maintain the aesthetics:
a + ggplot2::scale_fill_brewer(name = "Gender", labels = c("Male", "Female"))

# Clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.boxPlotGG` *Renders boxplot*

Description

Internal function. Renders a ggplot boxplot by retrieving from the server side a list with the identity stats and other parameters to render the plot without passing any data from the original dataset

Usage

```
ds.boxPlotGG(
  x,
  group = NULL,
  group2 = NULL,
  xlabel = "x axis",
  ylabel = "y axis",
  type = "pooled",
  datasources = NULL
)
```

Arguments

`x`	`character` Name on the server side of the data frame to form a boxplot. Structure on the server of this object must be: Column 'x': Names on the X axis of the boxplot, aka variables to plot Column 'value': Values for that variable (raw data of columns rbinded) Column 'group': (Optional) Values of the grouping variable Column 'group2': (Optional) Values of the second grouping variable
`group`	`character` (default `NULL`) Name of the first grouping variable.
`group2`	`character` (default `NULL`) Name of the second grouping variable.
`xlabel`	`caracter` (default `"x axis"`) Label to put on the x axis of the plot
`ylabel`	`caracter` (default `"y axis"`) Label to put on the y axis of the plot
`type`	`character` Return a pooled plot (`"pooled"`) or a split plot (one for each study server `"split"`)
`datasources`	a list of `DSConnection-class` (default `NULL`) objects obtained after login

Value

`ggplot` object

`ds.boxPlotGG_data_Treatment` *Take a data frame on the server side an arrange it to pass it to the boxplot function*

Description

Internal function

Usage

```
ds.boxPlotGG_data_Treatment(
  table,
  variables,
  group = NULL,
  group2 = NULL,
  datasources = NULL
)
```

Arguments

`table`	`character` Name of the table on the server side that holds the information to be plotted later
`variables`	`character vector` Name of the column(s) of the data frame to include on the boxplot
`group`	`character` (default `NULL`) Name of the first grouping variable.
`group2`	`character` (default `NULL`) Name of the second grouping variable.
`datasources`	a list of `DSConnection-class` (default `NULL`) objects obtained after login

Value

Does not return nothing, it creates the table `"boxPlotRawData"` on the server arranged to be passed to the ggplot boxplot function. Structure of the created table:

Column 'x': Names on the X axis of the boxplot, aka variables to plot
Column 'value': Values for that variable (raw data of columns rbinded)
Column 'group': (Optional) Values of the grouping variable
Column 'group2': (Optional) Values of the second grouping variable

ds.boxPlotGG_data_Treatment_numeric

Take a vector on the server side an arrange it to pass it to the boxplot function

Description

Internal function

Usage

```
ds.boxPlotGG_data_Treatment_numeric(vector, datasources = NULL)
```

Arguments

vector	character Name of the table on the server side that holds the information to be plotted later
datasources	a list of DSConnection-class (default NULL) objects obtained after login

Value

Does not return nothing, it creates the table "boxPlotRawDataNumeric" on the server arranged to be passed to the ggplot boxplot function. Structure of the created table:

Column 'x': Names on the X axis of the boxplot, aka name of the vector (vector argument)
Column 'value': Values for that variable

ds.boxPlotGG_numeric *Draw boxplot with information from a numeric vector*

Description

Draw boxplot with information from a numeric vector

Usage

```
ds.boxPlotGG_numeric(
  x,
  xlabel = "x axis",
  ylabel = "y axis",
  type = "pooled",
  datasources = NULL
)
```

Arguments

x	character Name of the numeric vector on the server side that holds the information to be plotted
xlabel	caracter (default "x axis") Label to put on the x axis of the plot
ylabel	caracter (default "y axis") Label to put on the y axis of the plot
type	character Return a pooled plot ("pooled") or a split plot (one for each study server "split")
datasources	a list of DSConnection-class (default NULL) objects obtained after login

Value

ggplot object

---

ds.boxPlotGG_table *Draw boxplot with information from a data frame*

Description

Draws a boxplot with the option of adding two grouping variables from data held on a table

Usage

```
ds.boxPlotGG_table(
  x,
  variables,
  group = NULL,
  group2 = NULL,
  xlabel = "x axis",
  ylabel = "y axis",
  type = "pooled",
  datasources = NULL
)
```

Arguments

x	character Name of the table on the server side that holds the information to be plotted
variables	character vector Name of the column(s) of the data frame to include on the boxplot
group	character (default NULL) Name of the first grouping variable.
group2	character (default NULL) Name of the second grouping variable.
xlabel	caracter (default "x axis") Label to put on the x axis of the plot
ylabel	caracter (default "y axis") Label to put on the y axis of the plot
type	character Return a pooled plot ("pooled") or a split plot (one for each study server "split")
datasources	a list of DSConnection-class (default NULL) objects obtained after login

Value

ggplot object

---

ds.bp_standards *Calculates Blood pressure z-scores*

Description

The function calculates blood pressure z-scores in two steps: Step 1. Calculates z-score of height according to CDC growth chart (Not the WHO growth chart!). Step 2. Calculates z-score of BP according to the fourth report on BP management, USA

Usage

```
ds.bp_standards(
  sex = NULL,
  age = NULL,
  height = NULL,
  bp = NULL,
  systolic = TRUE,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

sex	the name of the sex variable. The variable should be coded as 1 for males and 2 for females. If it is coded differently (e.g. 0/1), then you can use the ds.recodeValues function to recode the categories to 1/2 before the use of ds.bp_standards
age	the name of the age variable in years.
height	the name of the height variable in cm.
bp	the name of the blood pressure variable.
systolic	logical. If TRUE (default) the function assumes conversion of systolic blood pressure. If FALSE the function assumes conversion of diastolic blood pressure.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default name is set to bp.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Value

assigns a new object on the server-side. The assigned object is a list with two elements: the 'Zbp' which is the zscores of the blood pressure and 'perc' which is the percentiles of the BP zscores.

Author(s)

Demetris Avraam for DataSHIELD Development Team

References

The fourth report on the diagnosis, evaluation, and treatment of high blood pressure in children and adolescents: https://www.nhlbi.nih.gov/sites/default/files/media/docs/hbp_ped.pdf

---

ds.c *Combines values into a vector or list in the server-side*

Description

Concatenates objects into one vector.

Usage

```
ds.c(x = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x	a vector of character string providing the names of the objects to be combined.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default c.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

To avoid combining the character names and not the vectors on the client-side, the names are coerced into a list and the server-side function loops through that list to concatenate the list's elements into a vector.

Server function called: cDS

Value

ds.c returns the vector of concatenating R objects which are written to the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Create a vector with combined objects
  myvect <- c("D$LAB_TSC", "D$LAB_HDL")
  ds.c(x = myvect,
       newobj = "new.vect",
       datasources = connections[1]) #only the first Opal server is used ("study1")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.cbind` *Combines R objects by columns in the server-side*

Description

Takes a sequence of vector, matrix or data-frame arguments and combines them by column to produce a data-frame.

Usage

```
ds.cbind(
  x = NULL,
  DataSHIELD.checks = FALSE,
  force.colnames = NULL,
  newobj = NULL,
  datasources = NULL,
  notify.of.progress = FALSE
)
```

Arguments

x	a character vector with the name of the objects to be combined.
DataSHIELD.checks	logical. if TRUE does four checks: 1. the input object(s) is(are) defined in all the studies. 2. the input object(s) is(are) of the same legal class in all the studies. 3. if there are any duplicated column names in the input objects in each study. 4. the number of rows is the same in all components to be cbind. Default FALSE.
force.colnames	can be NULL (recommended) or a vector of characters that specifies column names of the output object. If it is not NULL the user should take some caution. For more information see **Details**.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Defaults cbind.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa
notify.of.progress	specifies if console output should be produced to indicate progress. Default FALSE.

Details

A sequence of vector, matrix or data-frame arguments is combined column by column to produce a data-frame that is written to the server-side.

This function is similar to the native R function cbind.

In DataSHIELD.checks the checks are relatively slow. Default DataSHIELD.checks value is FALSE.

If force.colnames is NULL (which is recommended), the column names are inferred from the names or column names of the first object specified in the x argument. If this argument is not NULL, then the column names of the assigned data.frame have the same order as the characters specified by the user in this argument. Therefore, the vector of force.colnames must have the same number of elements as the columns in the output object. In a multi-site DataSHIELD setting to use this argument, the user should make sure that each study has the same number of names and column names of the input elements specified in the x argument and in the same order in all the studies.

Server function called: cbindDS

Value

ds.cbind returns a data frame combining the columns of the R objects specified in the function which is written to the server-side. It also returns to the client-side two messages with the name of newobj that has been created in each data source and DataSHIELD.checks result.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Assign the exponent of a numeric variable at each server and cbind it
  # to the data frame D

  ds.exp(x = "D$LAB_HDL",
         newobj = "LAB_HDL.exp",
         datasources = connections)

  ds.cbind(x = c("D", "LAB_HDL.exp"),
           DataSHIELD.checks = FALSE,
           newobj = "D.cbind.1",
           datasources = connections)

 # Example 2: If there are duplicated column names in the input objects the function adds
 # a suffix '.k' to the kth replicate". If also the argument DataSHIELD.checks is set to TRUE
```

```
 # the function returns a warning message notifying the user for the existence of any duplicated
 # column names in each study

 ds.cbind(x = c("LAB_HDL.exp", "LAB_HDL.exp"),
          DataSHIELD.checks = TRUE,
          newobj = "D.cbind.2",
          datasources = connections)

 ds.colnames(x = "D.cbind.2",
             datasources = connections)

 # Example 3: Generate a random normally distributed variable of length 100 at each study,
 # and cbind it to the data frame D. This example fails and  returns an error as the length
# of the generated variable "norm.var" is not the same as the number of rows in the data frame D

 ds.rNorm(samp.size = 100,
          newobj = "norm.var",
          datasources = connections)

 ds.cbind(x = c("D", "norm.var"),
          DataSHIELD.checks = FALSE,
          newobj = "D.cbind.3",
          datasources = connections)

 # Clear the Datashield R sessions and logout
 datashield.logout(connections)

## End(Not run)
```

---

ds.changeRefGroup *Changes the reference level of a factor in the server-side*

Description

Change the reference level of a factor, by putting the reference group first.

This function is similar to R function `relevel`.

Usage

```
ds.changeRefGroup(
  x = NULL,
  ref = NULL,
  newobj = NULL,
  reorderByRef = FALSE,
  datasources = NULL
)
```

Arguments

x	a character string providing the name of the input vector of type factor.
ref	the reference level.
newobj	a character string that provides the name for the output object that is stored on the server-side. Default changerefgroup.newobj.
reorderByRef	logical, if TRUE the new vector should be ordered by the reference group (i.e. putting the reference group first). The default is to not re-order (see the reasons in the details).
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function allows the user to re-order the vector, putting the reference group first. It should be mentioned that by default the reference is the first level in the vector of levels. If the user chooses the re-order a warning is issued as this can introduce a mismatch of values if the vector is put back into a table that is not reordered in the same way. Such mismatch can render the results of operations on that table invalid.

Server function called: changeRefGroupDS

Value

ds.changeRefGroup returns a new vector with the specified level as a reference which is written to the server-side.

Author(s)

DataSHIELD Development Team

See Also

ds.cbind Combines objects column-wise.

ds.levels to obtain the levels (categories) of a vector of type factor.

ds.colnames to obtain the column names of a matrix or a data frame

ds.asMatrix to coerce an object into a matrix type.

ds.dim to obtain the dimensions of a matrix or a data frame.

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
builder <- DSI::newDSLoginBuilder()
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Changing the reference group in the server-side

  # Example 1: rename the categories and change the reference with re-ordering
    # print out the levels of the initial vector
    ds.levels(x= "D$PM_BMI_CATEGORICAL",
              datasources = connections)

    # define a vector with the new levels and recode the initial levels
    newNames <- c("normal", "overweight", "obesity")
    ds.recodeLevels(x = "D$PM_BMI_CATEGORICAL",
                    newCategories = newNames,
                    newobj = "bmi_new",
                    datasources = connections)

    # print out the levels of the new vector
    ds.levels(x = "bmi_new",
              datasources = connections)

    # Set the reference to "obesity" without changing the order (default)
    ds.changeRefGroup(x = "bmi_new",
                      ref = "obesity",
                      newobj = "bmi_ob",
                      datasources = connections)

   # print out the levels; the first listed level (i.e. the reference) is now 'obesity'
    ds.levels(x = "bmi_ob",
              datasources = connections)

  # Example 2: change the reference and re-order by the reference level
    # If re-ordering is sought, the action is completed but a warning is issued
    ds.recodeLevels(x = "D$PM_BMI_CATEGORICAL",
                    newCategories = newNames,
                    newobj = "bmi_new",
                   datasources = connections)
```

```
ds.changeRefGroup(x = "bmi_new",
                  ref = "obesity",
                  newobj = "bmi_ob",
                  reorderByRef = TRUE,
                  datasources = connections)


# Clear the Datashield R sessions and logout
datashield.logout(connections)

## End(Not run)
```

`ds.class` *Class of the R object in the server-side*

Description

Retrieves the class of an R object. This function is similar to the R function `class`.

Usage

```
ds.class(x = NULL, datasources = NULL)
```

Arguments

`x`	a character string providing the name of the input R object.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Same as the native R function `class`.

Server function called: `classDS`

Value

`ds.class` returns the type of the R object.

Author(s)

DataSHIELD Development Team

See Also

`ds.exists` to verify if an object is defined (exists) on the server-side.

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Getting the class of the R objects stored in the server-side
  ds.class(x = "D", #whole dataset
           datasources = connections[1]) #only the first server ("study1") is used

  ds.class(x = "D$LAB_TSC", #select a variable
           datasources = connections[1]) #only the first server ("study1") is used

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.colnames` *Produces column names of the R object in the server-side*

Description

Retrieves column names of an R object on the server-side. This function is similar to R function `colnames`.

Usage

```
ds.colnames(x = NULL, datasources = NULL)
```

Arguments

x	a character string providing the name of the input data frame or matrix.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The input is restricted to the object of type data.frame or matrix.

Server function called: colnamesDS

Value

ds.colnames returns the column names of the specified server-side data frame or matrix.

Author(s)

DataSHIELD Development Team

See Also

ds.dim to obtain the dimensions of a matrix or a data frame.

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")
```

```
  # Getting column names of the R objects stored in the server-side
  ds.colnames(x = "D",
              datasources = connections[1]) #only the first server ("study1") is used
  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.completeCases` *Identifies complete cases in server-side R objects*

Description

Selects complete cases of a data frame, matrix or vector that contain missing values.

Usage

```
ds.completeCases(x1 = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x1`	a character denoting the name of the input object which can be a data frame, matrix or vector.
`newobj`	a character string that provides the name for the complete-cases object that is stored on the data servers. If the user does not specify a name, then the function generates a name for the generated object that is the name of the input object with the suffix "_complete.cases"
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified, the default set of connections will be used: see `datashield.connections_defa`

Details

In the case of a data frame or matrix, `ds.completeCases` deletes all rows containing one or more missing values. However `ds.completeCases` in vectors only deletes the observation recorded as NA.

Server function called: `completeCasesDS`

Value

`ds.completeCases` generates a modified data frame, matrix or vector from which all rows containing at least one NA have been deleted. The output object is stored on the server-side. Only two validity messages are returned to the client-side indicating the name of the `newobj` that has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Select complete cases from different R objects

  ds.completeCases(x1 = "D", #data frames in the Opal servers
                             #(see above the connection to the Opal servers)
                   newobj = "D.completeCases", # name for the output object
                                               # that is stored in the Opal servers
                   datasources = connections)  # All Opal servers are used
                                        # (see above the connection to the Opal servers)

 ds.completeCases(x1 = "D$LAB_TSC", #vector (variable) of the data frames in the Opal servers
                                    #(see above the connection to the Opal servers)
                   newobj = "LAB_TSC.completeCases", #name for the output variable
                                                     #that is stored in the Opal servers
             datasources = connections[2]) #only the second Opal server is used ("study2")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.contourPlot *Generates a contour plot*

Description

It generates a contour plot of the pooled data or one plot for each dataset on the client-side.

Usage

```
ds.contourPlot(
  x = NULL,
  y = NULL,
  type = "combine",
  show = "all",
  numints = 20,
  method = "smallCellsRule",
  k = 3,
  noise = 0.25,
  datasources = NULL
)
```

Arguments

x	a character string providing the name of a numerical vector.
y	a character string providing the name of a numerical vector.
type	a character string that represents the type of graph to display. If type is set to 'combine', a combined contour plot displayed and if type is set to 'split', each contour is plotted separately.
show	a character that represents where the plot should focus. If show is set to 'all', the ranges of the variables are used as plot limits. If show is set to 'zoomed', the plot is zoomed to the region where the actual data are.
numints	number of intervals for a density grid object.
method	a character that defines which contour will be created. If method is set to 'smallCellsRule' (default), the contour plot of the actual variables is created but grids with low counts are replaced with grids with zero counts. If method is set to 'deterministic' the contour of the scaled centroids of each k nearest neighbour of the original variables is created, where the value of k is set by the user. If the method is set to 'probabilistic', then the contour of 'noisy' variables is generated.
k	the number of the nearest neighbours for which their centroid is calculated. For more information see details.
noise	the percentage of the initial variance that is used as the variance of the embedded noise if the argument method is set to 'probabilistic'. For more information see details.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The ds.contourPlot function first generates a density grid and uses it to plot the graph. The cells of the grid density matrix that hold a count of less than the filter set by DataSHIELD (usually 5) are

considered invalid and turned into 0 to avoid potential disclosure. A message is printed to inform the user about the number of invalid cells.

The ranges returned by each study and used in the process of getting the grid density matrix are not the exact minimum and maximum values but rather close approximates of the real minimum and maximum value. This was done to reduce the risk of potential disclosure.

In the k parameter the user can choose any value for k equal to or greater than the pre-specified threshold used as a disclosure control for this method and lower than the number of observations minus the value of this threshold. k default value is 3 (we suggest k to be equal to, or bigger than, 3). Note that the function fails if the user uses the default value but the study has set a bigger threshold. The value of k is used only if the argument method is set to 'deterministic'. Any value of k is ignored if the argument method is set to 'probabilistic' or 'smallCellsRule'.

In noise any value of noise is ignored if the argument method is set to 'deterministic' or 'smallCellsRule'. The user can choose any value for noise equal to or greater than the pre-specified threshold 'nfilter.noise'. Default noise value is 0.25. The added noise follows a normal distribution with zero mean and variance equal to a percentage of the initial variance of each input variable.

Server functions called: heatmapPlotDS, rangeDS and densityGridDS

Value

ds.contourPlot returns a contour plot to the client-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
```

```
logindata <- builder$build()

# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Generating contour plots

ds.contourPlot(x = "D$LAB_TSC",
               y = "D$LAB_HDL",
               type = "combine",
               show = "all",
               numints = 20,
               method = "smallCellsRule",
               k = 3,
               noise = 0.25,
               datasources = connections)

# clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

---

ds.cor *Calculates the correlation of R objects in the server-side*

Description

This function calculates the correlation of two variables or the correlation matrix for the variables of an input data frame.

Usage

```
ds.cor(x = NULL, y = NULL, type = "split", datasources = NULL)
```

Arguments

x	a character string providing the name of the input vector, data frame or matrix.
y	a character string providing the name of the input vector, data frame or matrix. Default NULL.
type	a character string that represents the type of analysis to carry out. This must be set to 'split' or 'combine'. Default 'split'. For more information see details.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

In addition to computing correlations; this function produces a table outlining the number of complete cases and a table outlining the number of missing values to allow the user to decide the 'relevance' of the correlation based on the number of complete cases included in the correlation calculations.

If the argument y is not NULL, the dimensions of the object have to be compatible with the argument x.

The function calculates the pairwise correlations based on casewise complete cases which means that it omits all the rows in the input data frame that include at least one cell with a missing value, before the calculation of correlations.

If type is set to 'split' (default), the correlation of two variables or the variance-correlation matrix of an input data frame and the number of complete cases and missing values are returned for every single study. If type is set to 'combine', the pooled correlation, the total number of complete cases and the total number of missing values aggregated from all the involved studies, are returned.

Server function called: corDS

Value

ds.cor returns a list containing the number of missing values in each variable, the number of missing variables casewise, the correlation matrix, the number of used complete cases. The function applies two disclosure controls. The first disclosure control checks that the number of variables is not bigger than a percentage of the individual-level records (the allowed percentage is pre-specified by the 'nfilter.glm'). The second disclosure control checks that none of them is dichotomous with a level having fewer counts than the pre-specified 'nfilter.tab' threshold.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
```

```
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Get the correlation matrix of two continuous variables
  ds.cor(x="D$LAB_TSC", y="D$LAB_TRIG", type="combine", datasources = connections)

  # Example 2: Get the correlation matrix of the variables in a dataframe
  ds.dataFrame(x=c("D$LAB_TSC", "D$LAB_TRIG", "D$LAB_HDL", "D$PM_BMI_CONTINUOUS"),
               newobj="D.new", check.names=FALSE, datasources=connections)
  ds.cor("D.new", type="combine", datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.corTest` *Tests for correlation between paired samples in the server-side*

Description

This is similar to the R stats function `cor.test`.

Usage

```
ds.corTest(
  x = NULL,
  y = NULL,
  method = "pearson",
  exact = NULL,
  conf.level = 0.95,
  type = "split",
  datasources = NULL
)
```

Arguments

`x`	a character string providing the name of a numerical vector.
`y`	a character string providing the name of a numerical vector.
`method`	a character string indicating which correlation coefficient is to be used for the test. One of "pearson", "kendall", or "spearman", can be abbreviated. Default is set to "pearson".

exact	a logical indicating whether an exact p-value should be computed. Used for Kendall's tau and Spearman's rho. See *Details* of R stats function cor.test for the meaning of NULL (the default).
conf.level	confidence level for the returned confidence interval. Currently only used for the Pearson product moment correlation coefficient if there are at least 4 complete pairs of observations. Default is set to 0.95.
type	a character string that represents the type of analysis to carry out. This must be set to 'split' or 'combine'. Default is set to 'split'. If type is set to "combine" then an approximated pooled correlation is estimated based on Fisher's z transformation.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Runs a two-sided correlation test between paired samples, using one of Pearson's product moment correlation coefficient, Kendall's tau or Spearman's rho. Server function called: corTestDS

Value

ds.corTest returns to the client-side the results of the correlation test.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # test for correlation
  ds.corTest(x = "D$LAB_TSC",
             y = "D$LAB_HDL",
             datasources = connections[1]) #Only first server is used ("study1")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.cov *Calculates the covariance of R objects in the server-side*

Description

This function calculates the covariance of two variables or the variance-covariance matrix for the variables of an input data frame.

Usage

```
ds.cov(
  x = NULL,
  y = NULL,
  naAction = "pairwise.complete",
  type = "split",
  datasources = NULL
)
```

Arguments

x	a character string providing the name of the input vector, data frame or matrix.
y	a character string providing the name of the input vector, data frame or matrix. Default NULL.
naAction	a character string giving a method for computing covariances in the presence of missing values. This must be set to 'casewise.complete' or 'pairwise.complete'. Default 'pairwise.complete'. For more information see details.
type	a character string that represents the type of analysis to carry out. This must be set to 'split' or 'combine'. Default 'split'. For more information see details.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

In addition to computing covariances; this function produces a table outlining the number of complete cases and a table outlining the number of missing values to allow for the user to decide about the 'relevance' of the covariance based on the number of complete cases included in the covariance calculations.

If the argument y is not NULL, the dimensions of the object have to be compatible with the argument x.

If naAction is set to 'casewise.complete', then the function omits all the rows in the whole data frame that include at least one cell with a missing value before the calculation of covariances. If naAction is set to 'pairwise.complete' (default), then the function divides the input data frame to subset data frames formed by each pair between two variables (all combinations are considered) and omits the rows with missing values at each pair separately and then calculates the covariances of those pairs.

If type is set to 'split' (default), the covariance of two variables or the variance-covariance matrix of an input data frame and the number of complete cases and missing values are returned for every single study. If type is set to 'combine', the pooled covariance, the total number of complete cases and the total number of missing values aggregated from all the involved studies, are returned.

Server function called: covDS

Value

ds.cov returns a list containing the number of missing values in each variable, the number of missing values casewise or pairwise depending on the argument naAction, the covariance matrix, the number of used complete cases and an error message which indicates whether or not the input variables pass the disclosure controls. The first disclosure control checks that the number of variables is not bigger than a percentage of the individual-level records (the allowed percentage is pre-specified by the 'nfilter.glm'). The second disclosure control checks that none of them is dichotomous with a level having fewer counts than the pre-specified 'nfilter.tab' threshold. If any of the input variables do not pass the disclosure controls then all the output values are replaced with NAs. If all the variables are valid and pass the controls, then the output matrices are returned and also an error message is returned but it is replaced by NA.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
```

```
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Calculate the covariance between two vectors
ds.assign(newobj='labhdl', toAssign='D$LAB_HDL', datasources = connections)
ds.assign(newobj='labtsc', toAssign='D$LAB_TSC', datasources = connections)
ds.assign(newobj='gender', toAssign='D$GENDER', datasources = connections)
ds.cov(x = 'labhdl',
       y = 'labtsc',
       naAction = 'pairwise.complete',
       type = 'combine',
       datasources = connections)
ds.cov(x = 'labhdl',
       y = 'gender',
       naAction = 'pairwise.complete',
       type = 'combine',
       datasources = connections[1]) #only the first Opal server is used ("study1")

# clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.dataFrame` *Generates a data frame object in the server-side*

Description

Creates a data frame from its elemental components: pre-existing data frames, single variables or matrices.

Usage

```
ds.dataFrame(
  x = NULL,
```

```
  row.names = NULL,
  check.rows = FALSE,
  check.names = TRUE,
  stringsAsFactors = TRUE,
  completeCases = FALSE,
  DataSHIELD.checks = FALSE,
  newobj = NULL,
  datasources = NULL,
  notify.of.progress = FALSE
)
```

Arguments

x	a character string that provides the name of the objects to be combined.
row.names	NULL, integer or character string that provides the row names of the output data frame.
check.rows	logical. If TRUE then the rows are checked for consistency of length and names. Default is FALSE.
check.names	logical. If TRUE the column names in the data frame are checked to ensure that is unique. Default is TRUE.
stringsAsFactors	logical. If true the character vectors are converted to factors. Default TRUE.
completeCases	logical. If TRUE rows with one or more missing values will be deleted from the output data frame. Default is FALSE.
DataSHIELD.checks	logical. Default FALSE. If TRUE undertakes all DataSHIELD checks (time-consuming) which are: 1. the input object(s) is(are) defined in all the studies 2. the input object(s) is(are) of the same legal class in all the studies 3. if there are any duplicated column names in the input objects in each study 4. the number of rows of the data frames or matrices and the length of all component variables are the same
newobj	a character string that provides the name for the output data frame that is stored on the data servers. Default dataframe.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa
notify.of.progress	specifies if console output should be produced to indicate progress. Default is FALSE.

Details

It creates a data frame by combining pre-existing data frames, matrices or variables.

The length of all component variables and the number of rows of the data frames or matrices must be the same. The output data frame will have the same number of rows.

Server functions called: classDS, colnamesDS, dataFrameDS

Value

ds.dataFrame returns the object specified by the newobj argument which is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the newobj that has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")

  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Create a new data frame
  ds.dataFrame(x = c("D$LAB_TSC","D$GENDER","D$PM_BMI_CATEGORICAL"),
               row.names = NULL,
               check.rows = FALSE,
               check.names = TRUE,
               stringsAsFactors = TRUE, #character variables are converted to a factor
               completeCases = TRUE, #only rows with not missing values are selected
               DataSHIELD.checks = FALSE,
               newobj = "df1",
              datasources = connections[1], #only the first Opal server is used ("study1")
               notify.of.progress = FALSE)
```

```
  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.dataFrameFill` *Creates missing values columns in the server-side*

Description

Adds extra columns with missing values in a data frame on the server-side.

Usage

```
ds.dataFrameFill(df.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`df.name`	a character string representing the name of the input data frame that will be filled with extra columns of missing values.
`newobj`	a character string that provides the name for the output data frame that is stored on the data servers. Default value is "dataframefill.newobj".
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function checks if the input data frames have the same variables (i.e. the same column names) in all of the used studies. When a study does not have some of the variables, the function generates those variables as vectors of missing values and combines them as columns to the input data frame. If any of the generated variables are of class factor, the function assigns to those the corresponding levels of the factors given from the studies where such factors exist.

Server function called: `dataFrameFillDS`

Value

`ds.dataFrameFill` returns the object specified by the `newobj` argument which is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the `newobj` that has been created in each data source and if it is in a valid form.

Author(s)

Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")

  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Create two data frames with one different column

  ds.dataFrame(x = c("D$LAB_TSC","D$LAB_TRIG","D$LAB_HDL",
                     "D$LAB_GLUC_ADJUSTED","D$PM_BMI_CONTINUOUS"),
               newobj = "df1",
               datasources = connections[1])

  ds.dataFrame(x = c("D$LAB_TSC","D$LAB_TRIG","D$LAB_HDL","D$LAB_GLUC_ADJUSTED"),
               newobj = "df1",
               datasources = connections[2])

  # Fill the data frame with NA columns

  ds.dataFrameFill(df.name = "df1",
                   newobj = "D.Fill",
                   datasources = connections[c(1,2)]) # Two servers are used


  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.dataFrameSort` *Sorts data frames in the server-side*

Description

Sorts a data frame using a specified sort key.

Usage

```
ds.dataFrameSort(
  df.name = NULL,
  sort.key.name = NULL,
  sort.descending = FALSE,
  sort.method = "default",
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`df.name`	a character string providing the name of the data frame to be sorted.
`sort.key.name`	a character string providing the name for the sort key.
`sort.descending`	logical, if TRUE the data frame will be sorted. by the sort key in descending order. Default = FALSE (sort order ascending).
`sort.method`	a character string that specifies the method to be used to sort the data frame. This can be set as `"alphabetic"`,`"a"` or `"numeric"`, `"n"`.
`newobj`	a character string that provides the name for the output data frame that is stored on the data servers. Default `dataframesort.newobj`. where `df.name` is the first argument of `ds.dataFrameSort()`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

It sorts a specified data.frame on the serverside using a sort key also on the server-side. The sort key can either sit in the data.frame or outside it. The sort key can be forced to be interpreted as alphabetic or numeric.

When a numeric vector is sorted alphabetically, the order can look confusing. For example, if we have a numeric vector to sort:
vector.2.sort = c(-192, 76, 841, NA, 1670, 163, 147, 101, -112, -231, -9, 119, 112, NA)

When sorting numbers in an ascending (default) manner, the largest negative numbers get ordered first leading up to the largest positive numbers and finally (by default in R) NAs being positioned at the end of the vector:

numeric.sort = c(-231, -192, -112, -9, 76, 101, 112, 119, 147, 163, 841, 1670, NA, NA)

Instead, if the same vector is sorted alphabetically the the resultant vector is:

alphabetic.sort = (-112, -192, -231, -9, 101, 112, 119, 147, 163, 1670, 76, 841, NA, NA)

Server function called: `dataFrameSortDS`.

Value

`ds.dataFrameSort` returns the sorted data frame is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the `newobj` which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Sorting the data frame
  ds.dataFrameSort(df.name = "D",
                   sort.key.name = "D$LAB_TSC",
                   sort.descending = TRUE,
                   sort.method = "numeric",
                   newobj = "df.sort",
```

```
                datasources = connections[1]) #only the first Opal server is used ("study1")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.dataFrameSubset *Sub-sets data frames in the server-side*

Description

Subsets a data frame by rows and/or by columns.

Usage

```
ds.dataFrameSubset(
  df.name = NULL,
  V1.name = NULL,
  V2.name = NULL,
  Boolean.operator = NULL,
  keep.cols = NULL,
  rm.cols = NULL,
  keep.NAs = NULL,
  newobj = NULL,
  datasources = NULL,
  notify.of.progress = FALSE
)
```

Arguments

df.name	a character string providing the name of the data frame to be subset.
V1.name	A character string specifying the name of the vector to which the Boolean operator is to be applied to define the subset. For more information see details.
V2.name	A character string specifying the name of the vector to compare with V1.name.
Boolean.operator	A character string specifying one of six possible Boolean operators: '==', '!=', '>', '>=', '<' and '<='.
keep.cols	a numeric vector specifying the numbers of the columns to be kept in the final subset.
rm.cols	a numeric vector specifying the numbers of the columns to be removed from the final subset.
keep.NAs	logical, if TRUE the missing values are included in the subset. If FALSE or NULL all rows with at least one missing values are removed from the subset.

newobj — a character string that provides the name for the output object that is stored on the data servers. Default dataframesubset.newobj.

datasources — a list of DSConnection-class objects obtained after login. If the datasources the default set of connections will be used: see datashield.connections_default.

notify.of.progress — specifies if console output should be produced to indicate progress. Default FALSE.

Details

Subset a pre-existing data frame using the standard set of Boolean operators (==, !=, >, >=, <, <=). The subsetting is made by rows, but it is also possible to select columns to keep or remove. Instead, if you wish to keep all rows in the subset (e.g. if the primary plan is to subset by columns and not by rows) the V1.name and V2.name parameters can be used to specify a vector of the same length as the data frame to be subsetted in each study in which every element is 1 and there are no missing values. For more information see the example 2 below.

Server functions called: dataFrameSubsetDS1 and dataFrameSubsetDS2

Value

ds.dataFrameSubset returns the object specified by the newobj argument which is written to the server-side. Also, two validity messages are returned to the client-side indicating the name of the newobj which has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
```

```
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Subsetting a data frame
  #Example 1: Include some rows and all columns in the subset
  ds.dataFrameSubset(df.name = "D",
                     V1.name = "D$LAB_TSC",
                     V2.name = "D$LAB_TRIG",
                     Boolean.operator = ">",
                     keep.cols = NULL, #All columns are included in the new subset
                     rm.cols = NULL, #All columns are included in the new subset
                     keep.NAs = FALSE, #All rows with NAs are removed
                     newobj = "new.subset",
                   datasources = connections[1],#only the first server is used ("study1")
                     notify.of.progress = FALSE)
  #Example 2: Include all rows and some columns in the new subset
    #Select complete cases (rows without NA)
    ds.completeCases(x1 = "D",
                     newobj = "complet",
                     datasources = connections)
    #Create a vector with all ones
    ds.make(toAssign = "complet$LAB_TSC-complet$LAB_TSC+1",
            newobj = "ONES",
            datasources = connections)
    #Subset the data
    ds.dataFrameSubset(df.name = "complet",
                       V1.name = "ONES",
                       V2.name = "ONES",
                       Boolean.operator = "==",
                      keep.cols = c(1:4,10), #only columns 1, 2, 3, 4 and 10 are selected
                       rm.cols = NULL,
                       keep.NAs = FALSE,
                       newobj = "subset.all.rows",
                       datasources = connections, #all servers are used
                       notify.of.progress = FALSE)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.densityGrid *Generates a density grid in the client-side*

Description

This function generates a grid density object which can then be used to produced heatmap or contour plots.

Usage

```
ds.densityGrid(
  x = NULL,
  y = NULL,
  numints = 20,
  type = "combine",
  datasources = NULL
)
```

Arguments

x	a character string providing the name of the input numerical vector.
y	a character string providing the name of the input numerical vector.
numints	an integer, the number of intervals for the grid density object. The default value is 20.
type	a character string that represents the type of graph to display. If type is set to 'combine', a pooled grid density matrix is generated, instead if type is set to 'split' one grid density matrix is generated. Default 'combine'.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The cells with a count > 0 and < nfilter.tab are considered invalid and the count is set to 0.

In DataSHIELD the user does not have access to the micro-data so and extreme values such as the maximum and the minimum are potentially non-disclosive so this function does not allow for the user to set the limits of the density grid and the minimum and maximum values of the x and y vectors. These elements are set by the server-side function densityGridDS to 'valid' values (i.e. values that do not lead to leakage of micro-data to the user).

Server function called: densityGridDS

Value

ds.densityGrid returns a grid density matrix.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Generate the density grid
  # Example1: generate a combined grid density object (default)
  ds.densityGrid(x="D$LAB_TSC",
                 y="D$LAB_HDL",
                 datasources = connections)#all opal servers are used

  # Example2: generate a grid density object for each study separately
  ds.densityGrid(x="D$LAB_TSC",
                 y="D$LAB_HDL",
                 type="split",
               datasources = connections[1])#only the first Opal server is used ("study1")

 # Example3: generate a grid density object where the number of intervals is set to 15, for
  #           each study separately
  ds.densityGrid(x="D$LAB_TSC",
                 y="D$LAB_HDL",
                 type="split",
                 numints=15,
                 datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)
```

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

---

ds.dim *Retrieves the dimension of a server-side R object*

Description

Gives the dimensions of an R object on the server-side. This function is similar to R function dim.

Usage

```
ds.dim(x = NULL, type = "both", checks = FALSE, datasources = NULL)
```

Arguments

x	a character string providing the name of the input object.
type	a character string that represents the type of analysis to carry out. If type is set to 'combine', 'combined', 'combines' or 'c', the global dimension is returned. If type is set to 'split', 'splits' or 's', the dimension is returned separately for each study. If type is set to 'both' or 'b', both sets of outputs are produced. Default 'both'.
checks	logical. If TRUE undertakes all DataSHIELD checks (time-consuming). Default FALSE.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The function returns the dimension of the server-side input object (e.g. array, matrix or data frame) from every single study and the pooled dimension of the object by summing up the individual dimensions returned from each study.

In checks parameter is suggested that checks should only be undertaken once the function call has failed.

Server function called: dimDS

Value

ds.dim retrieves to the client-side the dimension of the object in the form of a vector where the first element indicates the number of rows and the second element indicates the number of columns.

Author(s)

DataSHIELD Development Team

See Also

ds.dataFrame to generate a table of the type data frame.

ds.changeRefGroup to change the reference level of a factor.

ds.colnames to obtain the column names of a matrix or a data frame

ds.asMatrix to coerce an object into a matrix type.

ds.length to obtain the size of a vector.

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


  # Calculate the dimension
  ds.dim(x="D",
         type="combine", #global dimension
         checks = FALSE,
         datasources = connections)#all opal servers are used
  ds.dim(x="D",
         type = "both",#separate dimension for each study
                       #and the pooled dimension (default)
         checks = FALSE,
         datasources = connections)#all opal servers are used
  ds.dim(x="D",
         type="split", #separate dimension for each study
         checks = FALSE,
```

```
        datasources = connections[1])#only the first opal server is used ("study1")

  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.dmtC2S *Copy a clientside data.frame, matrix or tibble to the serverside*

Description

Creates a data.frame, matrix or tibble on the serverside that is equivalent to that same data.frame, matrix or tibble (DMT) on the clientside.

Usage

```
ds.dmtC2S(dfdata = NA, newobj = NULL, datasources = NULL)
```

Arguments

dfdata	is a character string that specifies the name of the DMT to be copied from the clientside to the serverside
newobj	A character string specifying the name of the DMT on the serverside to which the output is to be written. If no <newobj> argument is specified or it is NULL the name of the copied DMT defaults to "dmt.copied.C2S".
datasources	specifies the particular 'connection object(s)' to use. e.g. if you have several data sets in the sources you are working with called opals.a, opals.w2, and connection.xyz, you can choose which of these to work with. The call 'datashield.connections_find()' lists all of the different datasets available and if one of these is called 'default.connections' that will be the dataset used by default if no other dataset is specified. If you wish to change the connections you wish to use by default the call datashield.connections_default('opals.a') will set 'default.connections' to be 'opals.a' and so in the absence of specific instructions to the contrary (e.g. by specifying a particular dataset to be used via the <datasources> argument) all subsequent function calls will be to the datasets held in opals.a. If the <datasources> argument is specified, it should be set without inverted commas: e.g. datasources=opals.a or datasources=default.connections. The <datasources> argument also allows you to apply a function solely to a subset of the studies/sources you are working with. For example, the second source in a set of three, can be specified using a call such as datasources=connection.xyz[2]. On the other hand, if you wish to specify solely the first and third sources, the appropriate call will be datasources=connections.xyz[c(1,3)]

Details

ds.dmtC2S calls assign function dmtC2SDS. To keep the function simple (though less flexible), a number of the parameters specifying the DMT to be generated on the serverside are fixed by the characteristics of the DMT to be copied rather than explicitly specifying them as selected arguments. In consequence, they have been removed from the list of arguments and are instead given invariant values in the first few lines of code. These include: from="clientside.dmt", nrows.scalar=NULL, ncols.scalar=NULL, byrow = FALSE. The specific value "clientside.dmt" for the argument <from> simply means that the required information is generated from the characteristics of a clientside DMT. The <nrows.scalar> and <ncols.scalar> are fixed empirically by the number of rows and columns of the DMT to be copied. <byrow> specifies writing the serverside DMT by columns or by rows and this is defaulted to byrow=FALSE i.e. "by column".

Value

the object specified by the <newobj> argument (or default name "dmt.copied.C2S") which is written as a data.frame/matrix/tibble to the serverside.

Author(s)

Paul Burton for DataSHIELD Development Team - 3rd June, 2021

---

ds.elspline *Basis for a piecewise linear spline with meaningful coefficients*

Description

This function is based on the native R function `elspline` from the `lspline` package. This function computes the basis of piecewise-linear spline such that, depending on the argument marginal, the coefficients can be interpreted as (1) slopes of consecutive spline segments, or (2) slope change at consecutive knots.

Usage

```
ds.elspline(
  x,
  n,
  marginal = FALSE,
  names = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

x	the name of the input numeric variable
n	integer greater than 2, knots are computed such that they cut n equally-spaced intervals along the range of x

marginal	logical, how to parametrise the spline, see Details
names	character, vector of names for constructed variables
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default elspline.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

If marginal is FALSE (default) the coefficients of the spline correspond to slopes of the consecutive segments. If it is TRUE the first coefficient correspond to the slope of the first segment. The consecutive coefficients correspond to the change in slope as compared to the previous segment. Function elspline wraps lspline and computes the knot positions such that they cut the range of x into n equal-width intervals.

Value

an object of class "lspline" and "matrix", which its name is specified by the newobj argument (or its default name "elspline.newobj"), is assigned on the serverside.

Author(s)

Demetris Avraam for DataSHIELD Development Team

---

ds.exists *Checks if an object is defined on the server-side*

Description

Looks if an R object of the given name is defined on the server-side. This function is similar to the R function exists.

Usage

```
ds.exists(x = NULL, datasources = NULL)
```

Arguments

x	a character string providing the name of the object to look for.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

In DataSHIELD it is not possible to see the data on the servers of the collaborating studies. It is only possible to get summaries of objects stored on the server-side. It is however important to know if an object is defined (i.e. exists) on the server-side. This function checks if an object does exist on the server-side.

Server function called: exists

Value

`ds.exists` returns a logical object. TRUE if the object is on the server-side and FALSE otherwise.

Author(s)

DataSHIELD Development Team

See Also

`ds.class` to check the type of an object.

`ds.length` to check the length of an object.

`ds.dim` to check the dimension of an object.

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Check if the object exist in the server-side
  ds.exists(x = "D",
            datasources = connections) #All opal servers are used
  ds.exists(x = "D",
            datasources = connections[1]) #Only the first Opal server is used (study1)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)
```

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

---

ds.exp *Computes the exponentials in the server-side*

Description

Computes the exponential values for a specified numeric vector. This function is similar to R function exp.

Usage

```
ds.exp(x = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x	a character string providing the name of a numerical vector.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default exp.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Server function called: exp.

Value

ds.exp returns a vector for each study of the exponential values for the numeric vector specified in the argument x. The created vectors are stored in the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")

  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # compute exponential function of the 'PM_BMI_CONTINUOUS' variable
  ds.exp(x = "D$PM_BMI_CONTINUOUS",
         newobj = "exp.PM_BMI_CONTINUOUS",
         datasources = connections[1]) #only the first Opal server is used (study1)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.extractQuantiles` *Secure ranking of a vector across all sources and use of these ranks to estimate global quantiles across all studies*

Description

Takes the global ranks and quantiles held in the serverside data data frame that is written by ranksSecureDS4 and named as specified by the argument (<output.ranks.df>) and converts these values into a series of quantile values that identify, for example, which value of V2BR across all of the studies corresponds to the median or to the 95 indication in which study the V2BR corresponding to a particular quantile falls and, in fact, the relevant value may fall in more than one study and may appear multiple times in any one study. Finally, the output data frame containing this information is written to the clientside and to the serverside at each study separately.

Usage

```
ds.extractQuantiles(
  extract.quantiles,
```

```
  extract.summary.output.ranks.df,
  extract.ranks.sort.by,
  extract.rm.residual.objects,
  extract.datasources = NULL
)
```

Arguments

`extract.quantiles`
: one of a restricted set of character strings. The value of this argument is set in choosing the value of the argument <quantiles.for.estimation> in ds.ranksSecure. In summary: to mitigate disclosure risk only the following set of quantiles can be generated: c(0.025,0.05,0.10,0.20,0.25,0.30,0.3333,0.40,0.50,0.60,0.6667, 0.70,0.75,0.80,0.90,0.95,0.97 The allowable formats for the argument are of the general form: "0.025-0.975" where the first number is the lowest quantile to be estimated and the second number is the equivalent highest quantile to estimate. These two quantiles are then estimated along with all allowable quantiles in between. The allowable argument values are then: "0.025-0.975", "0.05-0.95", "0.10-0.90", "0.20-0.80". Two alternative values are "quartiles" i.e. c(0.25,0.50,0.75), and "median" i.e. c(0.50). The default value is "0.05-0.95". For more details, see the associated document "secure.global.ranking.docx". Also see the header file for ds.ranksSecure.

`extract.summary.output.ranks.df`
: a character string which specifies the optional name for the summary data.frame written to the serverside on each data source that contains 5 of the key output variables from the ranking procedure pertaining to that particular data source. If no name has been specified by the argument <summary.output.ranks.df> in ds.ranksSecure, the default name is allocated as "summary.ranks.df".The only reason the <extract.summary.output.ranks.df> argument needs specifying in ds.extractQuantiles is because, ds.extractQuantiles is the last function called by ds.ranksSecure and almost the final command of ds.extractQuantiles to print out the name of the data frame containing the summarised ranking information generated by ds.ranksSecure and the order in which the data frame is laid out. This therefore appears as the last output produced when ds.ranksSecure is run, and when this happens it is clear this relates to the main output of ds.ranksSecure not of ds.extractQuantiles.

`extract.ranks.sort.by`
: a character string taking two possible values. These are "ID.orig" and "vals.orig". This is set via the argument <ranks.sort.by> in ds.ranksSecure. For more details see the associated document entitled "secure.global.ranking.docx". Also see the header file for ds.ranksSecure.

`extract.rm.residual.objects`
: logical value. Default = TRUE: at the beginning and end of each run of ds.ranksSecure delete all extraneous objects that are otherwise left behind. These are not usually needed, but could be of value if one were investigating a problem with the ranking. FALSE: do not delete the residual objects

`extract.datasources`
: specifies the particular opal object(s) to use. This is set via the argument<datasources> in ds.ranksSecure. For more details see the associated document entitled "secure.global.ranking.docx". Also see the header file for ds.ranksSecure.

Details

ds.extractQuantiles is a clientside function which should usually be called from within the clientside function ds.ranksSecure.If you try to call ds.extractQuantiles directly(i.e. not by running ds.ranksSecure) you are almost certainly going to have to set up quite a few vectors and scalars that are normally set by ds.ranksSecure and this is likely to be difficult. ds.extractQuantiles itself calls two serverside functions extractQuantilesDS1 and extractQuantilesDS2. For more details about the cluster of functions that collectively enable secure global ranking and estimation of global quantiles see the associated document entitled "secure.global.ranking.docx". In particular this explains how ds.extractQuantiles works. Also see the header file for ds.ranksSecure.

Value

the final main output of ds.extractQuantiles is a data frame object named "final.quantile.df". This contains two vectors. The first named "evaluation.quantiles" lists the full set of quantiles you have requested for evaluation as specified by the argument "quantiles.for.estimation" in ds.ranksSecure and explained in more detail above under the information for the argument "extract.quantiles" in this function. The second vector is called "final.quantile.vector" which details the values of V2BR that correspond to the evaluation quantiles in vector 1. The information in the data frame "final.quantile.df" is generic: there is no information identifying in which study each value of V2BR falls. This data frame is written to the clientside (as it is non-disclosive) and is also copied to the serverside in every study. This means it is easily accessible from anywhere in the DataSHIELD environment. For more details see the associated document entitled "secure.global.ranking.docx".

Author(s)

Paul Burton 11th November, 2021

---

ds.forestplot *Forestplot for SLMA models*

Description

Draws a forestplot of the coefficients for Study-Level Meta-Analysis performed with DataSHIELD

Usage

```
ds.forestplot(mod, variable = NULL, method = "ML", layout = "JAMA")
```

Arguments

mod	list List outputted by any of the SLMA models of DataSHIELD (ds.glmerSLMA, ds.glmSLMA, ds.lmerSLMA)
variable	character (default NULL) Variable to meta-analyse and visualise, by setting this argument to NULL (default) the first independent variable will be used.
method	character (Default "ML") Method to estimate the between study variance. See details from ?meta::metagen for the different options.
layout	character (default "JAMA") Layout of the plot. See details from ?meta::metagen for the different options.

Value

Results a foresplot object created with ‘meta::forest‘.

Examples

```
## Not run:
  # Run a logistic regression

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Fit the logistic regression model

  mod <- ds.glmSLMA(formula = "DIS_DIAB~GENDER+PM_BMI_CONTINUOUS+LAB_HDL",
                data = "D",
                family = "binomial",
                datasources = connections)

  # Plot the results of the model
  ds.forestplot(mod)

## End(Not run)
```

`ds.gamlss` *Generalized Additive Models for Location Scale and Shape*

Description

This function calls the gamlssDS that is a wrapper function from the gamlss R package. The function returns an object of class "gamlss", which is a generalized additive model for location, scale and shape (GAMLSS). The function also saves the residuals as an object on the server-side with a name specified by the newobj argument. In addition, if the argument centiles is set to TRUE, the function calls the centiles function from the gamlss package and returns the sample percentages below each centile curve.

Usage

```
ds.gamlss(
  formula = NULL,
  sigma.formula = "~1",
  nu.formula = "~1",
  tau.formula = "~1",
  family = "NO()",
  data = NULL,
  method = "RS",
  mu.fix = FALSE,
  sigma.fix = FALSE,
  nu.fix = FALSE,
  tau.fix = FALSE,
  control = c(0.001, 20, 1, 1, 1, 1, Inf),
  i.control = c(0.001, 50, 30, 0.001),
  centiles = FALSE,
  xvar = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

formula	a formula object, with the response on the left of an ~ operator, and the terms, separated by + operators, on the right. Nonparametric smoothing terms are indicated by pb() for penalised beta splines, cs for smoothing splines, lo for loess smooth terms and random or ra for random terms, e.g. 'y~cs(x,df=5)+x1+x2*x3'.
sigma.formula	a formula object for fitting a model to the sigma parameter, as in the formula above, e.g. sigma.formula='~cs(x,df=5)'.
nu.formula	a formula object for fitting a model to the nu parameter, e.g. nu.formula='~x'.
tau.formula	a formula object for fitting a model to the tau parameter, e.g. tau.formula='~cs(x,df=2)'.
family	a gamlss.family object, which is used to define the distribution and the link functions of the various parameters. The distribution families supported by gamlss() can be found in gamlss.family. Functions such as 'BI()' (binomial) produce a family object. Also can be given without the parentheses i.e. 'BI'. Family functions can take arguments, as in 'BI(mu.link=probit)'.
data	a data frame containing the variables occurring in the formula. If this is missing, the variables should be on the parent environment.
method	a character indicating the algorithm for GAMLSS. Can be either 'RS', 'CG' or 'mixed'. If method='RS' the function will use the Rigby and Stasinopoulos algorithm, if method='CG' the function will use the Cole and Green algorithm, and if method='mixed' the function will use the RS algorithm twice before switching to the Cole and Green algorithm for up to 10 extra iterations.
mu.fix	logical, indicate whether the mu parameter should be kept fixed in the fitting processes.

sigma.fix	logical, indicate whether the sigma parameter should be kept fixed in the fitting processes.
nu.fix	logical, indicate whether the nu parameter should be kept fixed in the fitting processes.
tau.fix	logical, indicate whether the tau parameter should be kept fixed in the fitting processes.
control	this sets the control parameters of the outer iterations algorithm using the gamlss.control function. This is a vector of 7 numeric values: (i) c.crit (the convergence criterion for the algorithm), (ii) n.cyc (the number of cycles of the algorithm), (iii) mu.step (the step length for the parameter mu), (iv) sigma.step (the step length for the parameter sigma), (v) nu.step (the step length for the parameter nu), (vi) tau.step (the step length for the parameter tau), (vii) gd.tol (global deviance tolerance level). The default values for these 7 parameters are set to c(0.001, 20, 1, 1, 1, 1, Inf).
i.control	this sets the control parameters of the inner iterations of the RS algorithm using the glim.control function. This is a vector of 4 numeric values: (i) cc (the convergence criterion for the algorithm), (ii) cyc (the number of cycles of the algorithm), (iii) bf.cyc (the number of cycles of the backfitting algorithm), (iv) bf.tol (the convergence criterion (tolerance level) for the backfitting algorithm). The default values for these 4 parameters are set to c(0.001, 50, 30, 0.001).
centiles	logical, indicating whether the function centiles() will be used to tabulate the sample percentages below each centile curve. Default is set to FALSE.
xvar	the unique explanatory variable used in the centiles() function. This variable is used only if the centiles argument is set to TRUE. A restriction in the centiles function is that it applies to models with one explanatory variable only.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default gamlss_res.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

For additional details see the help header of gamlss and centiles functions in native R gamlss package.

Value

a gamlss object with all components as in the native R gamlss function. Individual-level information like the components y (the response response) and residuals (the normalised quantile residuals of the model) are not disclosed to the client-side.

Author(s)

Demetris Avraam for DataSHIELD Development Team

ds.getWGSR *Computes the WHO Growth Reference z-scores of anthropometric data*

Description

Calculate WHO Growth Reference z-score for a given anthropometric measurement This function is similar to R function `getWGSR` from the `zscorer` package.

Usage

```
ds.getWGSR(
  sex = NULL,
  firstPart = NULL,
  secondPart = NULL,
  index = NULL,
  standing = NA,
  thirdPart = NA,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`sex`	the name of the binary variable that indicates the sex of the subject. This must be coded as 1 = male and 2 = female. If in your project the variable sex has different levels, you should recode the levels to 1 for males and 2 for females using the `ds.recodeValues` DataSHIELD function before the use of the `ds.getWGSR`.
`firstPart`	Name of variable specifying: Weight (kg) for BMI/A, W/A, W/H, or W/L Head circumference (cm) for HC/A Height (cm) for H/A Length (cm) for L/A MUAC (cm) for MUAC/A Sub-scapular skinfold (mm) for SSF/A Triceps skinfold (mm) for TSF/A Give a quoted variable name as in (e.g.) "weight". Be careful with units (weight in kg; height, length, head circumference, and MUAC in cm; skinfolds in mm).
`secondPart`	Name of variable specifying: Age (days) for H/A, HC/A, L/A, MUAC/A, SSF/A, or TSF/A Height (cm) for BMI/A, or W/H Length (cm) for W/L Give a quoted variable name as in (e.g.) "age". Be careful with units (age in days; height and length in cm).
`index`	The index to be calculated and added to data. One of: bfa BMI for age

hca Head circumference for age
hfa Height for age
lfa Length for age
mfa MUAC for age
ssa Sub-scapular skinfold for age
tsa Triceps skinfold for age
wfa Weight for age
wfh Weight for height
wfl Weight for length
Give a quoted index name as in (e.g.) "wfh".

`standing` Variable specifying how stature was measured. If NA (default) then age (for "hfa" or "lfa") or height rules (for "wfh" or "wfl") will be applied. This must be coded as 1 = Standing; 2 = Supine; 3 = Unknown. Missing values will be recoded to 3 = Unknown. Give a single value (e.g."1"). If no value is specified then height and age rules will be applied.

`thirdPart` Name of variable specifying age (in days) for BMI/A. Give a quoted variable name as in (e.g.) "age". Be careful with units (age in days). If age is given in different units you should convert it in age in days using the `ds.make` DataSHIELD function before the use of the `ds.getWGSR`. For example if age is given in months then the transformation is given by the formula $age_days=age_months*(365.25/12)$.

`newobj` a character string that provides the name for the output variable that is stored on the data servers. Defaults `getWGSR.newobj`.

`datasources` a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The function calls the server-side function `getWGSRDS` that computes the WHO Growth Reference z-scores of anthropometric data for weight, height or length, MUAC (middle upper arm circumference), head circumference, sub-scapular skinfold and triceps skinfold. Note that the function might fail or return NAs when the variables are outside the ranges given in the WGS (WHO Child Growth Standards) reference (i.e. 45 to 120 cm for height and 0 to 60 months for age). It is up to the user to check the ranges and the units of their data.

Value

`ds.getWGSR` assigns a vector for each study that includes the z-scores for the specified index. The created vectors are stored in the servers.

Author(s)

Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # Connecting to the Opal servers
```

```
require('DSI')
require('DSOpal')
require('dsBaseClient')

builder <- DSI::newDSLoginBuilder()
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "ANTHRO.anthro1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "ANTHRO.anthro2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "ANTHRO.anthro3", driver = "OpalDriver")

logindata <- builder$build()

# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Example 1: Generate the weight-for-height (wfh) index
ds.getWGSR(sex = "D$sex", firstPart = "D$weight", secondPart = "D$height",
         index = "wfh", newobj = "wfh_index", datasources = connections)

# Example 2: Generate the BMI for age (bfa) index
ds.getWGSR(sex = "D$sex", firstPart = "D$weight", secondPart = "D$height",
      index = "bfa", thirdPart = "D$age", newobj = "bfa_index", datasources = connections)

# clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.glm` *Fits Generalized Linear Model*

Description

Fits a Generalized Linear Model (GLM) on data from single or multiple sources on the server-side.

Usage

```
ds.glm(
  formula = NULL,
```

```
  data = NULL,
  family = NULL,
  offset = NULL,
  weights = NULL,
  checks = FALSE,
  maxit = 20,
  CI = 0.95,
  viewIter = FALSE,
  viewVarCov = FALSE,
  viewCor = FALSE,
  datasources = NULL
)
```

Arguments

formula	an object of class formula describing the model to be fitted. For more information see **Details**.
data	a character string specifying the name of an (optional) data frame that contains all of the variables in the GLM formula.
family	identifies the error distribution function to use in the model. This can be set as "gaussian", "binomial" and "poisson". For more information see **Details**.
offset	a character string specifying the name of a variable to be used as an offset. ds.glm does not allow an offset vector to be written directly into the GLM formula. For more information see **Details**.
weights	a character string specifying the name of a variable containing prior regression weights for the fitting process. ds.glm does not allow a weights vector to be written directly into the GLM formula.
checks	logical. If TRUE ds.glm checks the structural integrity of the model. Default FALSE. For more information see **Details**.
maxit	a numeric scalar denoting the maximum number of iterations that are permitted before ds.glm declares that the model has failed to converge.
CI	a numeric value specifying the confidence interval. Default 0.95.
viewIter	logical. If TRUE the results of the intermediate iterations are printed. If FALSE only final results are shown. Default FALSE.
viewVarCov	logical. If TRUE the variance-covariance matrix of parameter estimates is returned. Default FALSE.
viewCor	logical. If TRUE the correlation matrix of parameter estimates is returned. Default FALSE.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Fits a GLM on data from a single source or multiple sources on the server-side. In the latter case, the data are co-analysed (when using ds.glm) by using an approach that is mathematically equivalent to placing all individual-level data from all sources in one central warehouse and analysing those

data using the conventional glm() function in R. In this situation marked heterogeneity between sources should be corrected (where possible) with fixed effects. For example, if each study in a (binary) logistic regression analysis has an independent intercept, it is equivalent to allowing each study to have a different baseline risk of disease. This may also be viewed as being an IP (individual person) meta-analysis with fixed effects.

In formula most shortcut notation for formulas allowed under R's standard glm() function is also allowed by ds.glm.

Many GLMs can be fitted very simply using a formula such as:

$y\ a + b + c + d$

which simply means fit a GLM with y as the outcome variable and a, b, c and d as covariates. By default all such models also include an intercept (regression constant) term.

Instead, if you need to fit a more complex model, for example:

$EVENT\ 1 + TID + SEXF * AGE.60$

In the above model the outcome variable is EVENT and the covariates TID (factor variable with level values between 1 and 6 denoting the period time), SEXF (factor variable denoting sex) and AGE.60 (quantitative variable representing age-60 in years). The term 1 forces the model to include an intercept term, in contrast if you use the term 0 the intercept term is removed. The * symbol between SEXF and AGE.60 means fit all possible main effects and interactions for and between those two covariates. This takes the value 0 in all males 0 * AGE.60 and in females 1 * AGE.60. This model is in example 1 of the section **Examples**. In this case the logarithm of the survival time is added as an offset (log(survtime)).

In the family argument can be specified three types of models to fit:

"gaussian" : conventional linear model with normally distributed errors

"binomial" : conventional unconditional logistic regression model

"poisson" : Poisson regression model which is the most used in survival analysis. The model used Piecewise Exponential Regression (PER) which typically closely approximates Cox regression in its main estimates and standard errors.

At present the gaussian family is automatically coupled with an identity link function, the binomial family with a logistic link function and the poisson family with a log link function.

The data argument avoids you having to specify the name of the data frame in front of each covariate in the formula. For example, if the data frame is called DataFrame you avoid having to write: $DataFrame\$y\ DataFrame\$a + DataFrame\$b + DataFrame\$c + DataFrame\$d$

The checks argument verifies that the variables in the model are all defined (exist) on the server-side at every study and that they have the correct characteristics required to fit the model. It is suggested to make checks argument TRUE if an unexplained problem in the model fit is encountered because the running process takes several minutes.

In maxit Logistic regression and Poisson regression models can require many iterations, particularly if the starting value of the regression constant is far away from its actual value that the GLM is trying to estimate. In consequence we often set maxit=30 but depending on the nature of the models you wish to fit, you may wish to be alerted much more quickly than this if there is a delay in convergence, or you may wish to all more iterations.

Privacy protected iterative fitting of a GLM is explained here:

(1) Begin with a guess for the coefficient vector to start iteration 1 (let's call it beta.vector[1]). Using beta.vector[1], run iteration 1 with each source calculating the resultant score vector (and information matrix) generated by its data - given beta.vector[1] - as the sum of the score vector components (and the sum of the components of the information matrix) derived from each individual data record in that source. NB in most models the starting values in beta.vector[1] are set to be zero for all parameters.

(2) Transmit the resultant score vector and information matrix from each source back to the clientside server (CS) at the analysis centre. Let's denote SCORE[1][j] and INFORMATION.MATRIX[1][j] as the score vector and information matrix generated by study j at the end of the 1st iteration.

(3) CS sums the score vectors, and equivalently the information matrices, across all studies (i.e. j = 1:S, where S is the number of studies). Note that, given beta.vector[1], this gives precisely the same final sums for the score vectors and information matrices as would have been obtained if all data had been in one central warehoused database and the overall score vector and information matrix at the end of the first iteration had been calculated (as is standard) by simply summing across all individuals. The only difference is that instead of directly adding all values across all individuals, we first sum across all individuals in each data source and then sum those study totals across all studies - i.e. this generates the same ultimate sums

(4) CS then calculates sum(SCORES)%*% inverse(sum(INFORMATION.MATRICES)) - heuristically this may be viewed as being "the sum of the score vectors divided (NB 'matrix division') by the sum of the information matrices". If one uses the conventional algorithm (IRLS) to update generalized linear models from iteration to iteration this quantity happens to be precisely the vector to be added to the current value of beta.vector (i.e. beta.vector[1]) to obtain beta.vector[2] which is the improved estimate of the beta.vector to be used in iteration 2. This updating algorithm is often called the IRLS (Iterative Reweighted Least Squares) algorithm - which is closely related to the Newton Raphson approach but uses the expected information rather than the observed information.

(5) Repeat steps (2)-(4) until the model converges (using the standard R convergence criterion). NB An alternative way to coherently pool the glm across multiple sources is to fit each glm to completion (i.e. multiple iterations until convergence) in each source and then return the final parameter estimates and standard errors to the CS where they could be pooled using study-level meta-analysis. An alternative function ds.glmSLMA allows you to do this. It will fit the glms to completion in each source and return the final estimates and standard errors (rather than score vectors and information matrices). It will then rely on functions in the R package metafor to meta-analyse the key parameters.

Server functions called: glmDS1 and glmDS2

Value

Many of the elements of the output list returned by ds.glm are equivalent to those returned by the glm() function in native R. However, potentially disclosive elements such as individual-level residuals and linear predictor values are blocked. In this case, only non-disclosive elements are returned from each study separately.

The list of elements returned by ds.glm is mentioned below:

Nvalid: total number of valid observational units across all studies.

Nmissing: total number of observational units across all studies with at least one data item missing.

Ntotal: total of observational units across all studies, the sum of valid and missing units.

disclosure.risk: risk of disclosure, the value 1 indicates that one of the disclosure traps has been triggered in that study.

errorMessage: explanation for any errors or disclosure risks identified.

nsubs: total number of observational units used by ds.glm function. nb usually is the same as nvalid.

iter: total number of iterations before convergence achieved.

family: error family and link function.

formula: model formula, see description of formula as an input parameter (above).

coefficients: a matrix with 5 columns:

First : the names of all of the regression parameters (coefficients) in the model

second : the estimated values

third : corresponding standard errors of the estimated values

fourth : the ratio of estimate/standard error.

fifth : the p-value treating that as a standardised normal deviate

dev: residual deviance.

df: residual degrees of freedom. nb residual degrees of freedom + number of parameters in model = nsubs.

output.information: reminder to the user that there is more information at the top of the output.

Also, the estimated coefficients and standard errors expanded with estimated confidence intervals with % coverage specified by ci argument are returned. For the poisson model, the output is generated on the scale of the linear predictor (log rates and log rate ratios) and the natural scale after exponentiation (rates and rate ratios).

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  # Example 1: Fitting GLM for survival analysis
  # For this analysis we need to load survival data from the server

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
               table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
logindata <- builder$build()

# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Fit the GLM

# make sure that the outcome is numeric
ds.asNumeric(x.name = "D$cens",
             newobj = "EVENT",
             datasources = connections)

# convert time id variable to a factor

ds.asFactor(input.var.name = "D$time.id",
            newobj = "TID",
            datasources = connections)

# create in the server-side the log(survtime) variable

ds.log(x = "D$survtime",
       newobj = "log.surv",
       datasources = connections)

ds.glm(formula = EVENT ~ 1 + TID + female * age.60,
       data = "D",
       family = "poisson",
       offset = "log.surv",
       weights = NULL,
       checks = FALSE,
       maxit = 20,
       CI = 0.95,
       viewIter = FALSE,
       viewVarCov = FALSE,
       viewCor = FALSE,
       datasources = connections)

# Clear the Datashield R sessions and logout
datashield.logout(connections)

# Example 2: run a logistic regression without interaction
# For this example we are going to load another dataset

builder <- DSI::newDSLoginBuilder()
```

```
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Fit the logistic regression model

  mod <- ds.glm(formula = "DIS_DIAB~GENDER+PM_BMI_CONTINUOUS+LAB_HDL",
                data = "D",
                family = "binomial",
                datasources = connections)

  mod #visualize the results of the model

# Example 3: fit a standard Gaussian linear model with an interaction
# We are using the same data as in example 2.

mod <- ds.glm(formula = "PM_BMI_CONTINUOUS~DIS_DIAB*GENDER+LAB_HDL",
              data = "D",
              family = "gaussian",
              datasources = connections)
mod

# Clear the Datashield R sessions and logout
datashield.logout(connections)

## End(Not run)
```

---

ds.glmerSLMA *Fits Generalized Linear Mixed-Effect Models via Study-Level Meta-Analysis*

Description

ds.glmerSLMA fits a Generalized Linear Mixed-Effects Model (GLME) on data from one or multiple sources with pooling via SLMA (study-level meta-analysis).

Usage

```
ds.glmerSLMA(
  formula = NULL,
  offset = NULL,
  weights = NULL,
  combine.with.metafor = TRUE,
  dataName = NULL,
  checks = FALSE,
  datasources = NULL,
  family = NULL,
  control_type = NULL,
  control_value = NULL,
  nAGQ = 1L,
  verbose = 0,
  start_theta = NULL,
  start_fixef = NULL,
  notify.of.progress = FALSE,
  assign = FALSE,
  newobj = NULL
)
```

Arguments

formula	an object of class formula describing the model to be fitted. For more information see **Details**.
offset	a character string specifying the name of a variable to be used as an offset.
weights	a character string specifying the name of a variable containing prior regression weights for the fitting process.
combine.with.metafor	logical. If TRUE the estimates and standard errors for each regression coefficient are pooled across studies using random-effects meta-analysis under maximum likelihood (ML), restricted maximum likelihood (REML) or fixed-effects meta-analysis (FE). Default TRUE.
dataName	a character string specifying the name of a data frame that contains all of the variables in the GLME formula. For more information see **Details**.
checks	logical. If TRUE ds.glmerSLMA checks the structural integrity of the model. Default FALSE. For more information see **Details**.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa
family	a character string specifying the distribution of the observed value of the outcome variable around the predictions generated by the linear predictor. This can be set as "binomial" or "poisson". For more information see **Details**.
control_type	an optional character string vector specifying the nature of a parameter (or parameters) to be modified in the convergence control options which can be viewed or modified via the glmerControl function of the package lme4. For more information see **Details**.

`control_value` numeric representing the new value which you want to allocate the control parameter corresponding to the `control-type`. For more information see **Details**.

`nAGQ` an integer value indicating the number of points per axis for evaluating the adaptive Gauss-Hermite approximation to the log-likelihood. Defaults 1, corresponding to the Laplace approximation. For more information see R `glmer` function help.

`verbose` an integer value. If $verbose > 0$ the output is generated during the optimization of the parameter estimates. If $verbose > 1$ the output is generated during the individual penalized iteratively reweighted least squares (PIRLS) steps. Default `verbose` value is 0 which means no additional output.

`start_theta` a numeric vector of length equal to the number of random effects. Specify to retain more control over the optimisation. See `glmer()` for more details.

`start_fixef` a numeric vector of length equal to the number of fixed effects (NB including the intercept). Specify to retain more control over the optimisation. See `glmer()` for more details.

`notify.of.progress` specifies if console output should be produced to indicate progress. Default FALSE.

`assign` a logical, indicates whether the function will call a second server-side function (an assign) in order to save the regression outcomes (i.e. a glmerMod object) on each server. Default FALSE.

`newobj` a character string specifying the name of the object to which the glmerMod object representing the model fit on the serverside in each study is to be written. This argument is used only when the argument `assign` is set to TRUE. If no <newobj> argument is specified, the output object defaults to "new.glmer.obj".

Details

`ds.glmerSLMA` fits a generalized linear mixed-effects model (GLME) - e.g. a logistic or Poisson regression model including both fixed and random effects - on data from single or multiple sources.

This function is similar to `glmer` function from `lme4` package in native R.

When there are multiple data sources, the GLME is fitted to convergence in each data source independently. The estimates and standard errors returned to the client-side which enable cross-study pooling using Study-Level Meta-Analysis (SLMA). The SLMA used by default `metafor` package but as the SLMA occurs on the client-side (a standard R environment), the user can choose any approach to meta-analysis. Additional information about fitting GLMEs using `glmer` function can be obtained using R help for `glmer` and the `lme4` package.

In `formula` most shortcut notation allowed by `glmer()` function is also allowed by `ds.glmerSLMA`. Many GLMEs can be fitted very simply using a formula like: $y\ a + b + (1|c)$ which simply means fit an GLME with `y` as the outcome variable (e.g. a binary case-control using a logistic regression model or a count or a survival time using a Poisson regression model), `a` and `b` as fixed effects, and `c` as a random effect or grouping factor.

It is also possible to fit models with random slopes by specifying a model such as $y\ a + b + (1 + b|c)$ where the effect of `b` can vary randomly between groups defined by `c`. Implicit nesting can be specified with formulas such as: $y\ a + b + (1|c/d)$ or $y\ a + b + (1|c) + (1|c : d)$.

The dataName argument avoids you having to specify the name of the data frame in front of each covariate in the formula. For example, if the data frame is called DataFrame you avoid having to write: $DataFrame\$y\ DataFrame\$a + DataFrame\$b + (1|DataFrame\$c)$.

The checks argument verifies that the variables in the model are all defined (exist) on the server-site at every study and that they have the correct characteristics required to fit the model. It is suggested to make checks argument TRUE if an unexplained problem in the model fit is encountered because the running process takes several minutes.

In the family argument can be specified two types of models to fit:

"binomial" : logistic regression models

"poisson" : poisson regression models

Note if you are fitting a gaussian model (a standard linear mixed model) you should use ds.lmerSLMA and not ds.glmerSLMA. For more information you can see R help for lmer and glmer.

In control_type at present only one such parameter can be modified, namely the tolerance of the convergence criterion to the gradient of the log-likelihood at the maximum likelihood achieved. We have enabled this because our practical experience suggests that in situations where the model looks to have converged with sensible parameter values but formal convergence is not being declared if we allow the model to be more tolerant to a non-zero gradient the same parameter values are obtained but formal convergence is declared. The default value for the check.conv.grad is 0.001 (note that the default value of this argument in ds.lmerSLMA is 0.002).

In control_value at present (see control_type) the only parameter this can be is the convergence tolerance check.conv.grad. In general, models will be identified as having converged more readily if the value set for check.conv.grad is increased from its default value (0.001). Please note that the risk of doing this is that the model is also more likely to be declared as having converged at a local maximum that is not the global maximum likelihood. This will not generally be a problem if the likelihood surface is well behaved but if you have a problem with convergence you might usefully compare all the parameter estimates and standard errors obtained using the default tolerance (0.001) even though that has not formally converged with those obtained after convergence using the higher tolerance.

Server function called: glmerSLMADS2

Value

Many of the elements of the output list returned by ds.glmerSLMA are equivalent to those returned by the glmer() function in native R. However, potentially disclosive elements such as individual-level residuals and linear predictor values are blocked. In this case, only non-disclosive elements are returned from each study separately.

The list of elements returned by ds.glmerSLMA is mentioned below:

coefficients: a matrix with 5 columns:

First : the names of all of the regression parameters (coefficients) in the model

second : the estimated values

third : corresponding standard errors of the estimated values

fourth : the ratio of estimate/standard error

fifth : the p-value treating that as a standardised normal deviate

CorrMatrix: the correlation matrix of parameter estimates.

VarCovMatrix: the variance-covariance matrix of parameter estimates.

weights: the vector (if any) holding regression weights.

offset: the vector (if any) holding an offset.

cov.scaled: equivalent to VarCovMatrix.

Nmissing: the number of missing observations in the given study.

Nvalid: the number of valid (non-missing) observations in the given study.

Ntotal: the total number of observations in the given study (Nvalid + Nmissing).

data: equivalent to input parameter dataName (above).

call: summary of key elements of the call to fit the model.

Once the study-specific output has been returned, the function returns the number of elements relating to the pooling of estimates across studies via study-level meta-analysis. These are as follows:

input.beta.matrix.for.SLMA: a matrix containing the vector of coefficient estimates from each study.

input.se.matrix.for.SLMA: a matrix containing the vector of standard error estimates for coefficients from each study.

SLMA.pooled.estimates: a matrix containing pooled estimates for each regression coefficient across all studies with pooling under SLMA via random-effects meta-analysis under maximum likelihood (ML), restricted maximum likelihood (REML) or via fixed-effects meta-analysis (FE).

convergence.error.message: reports for each study whether the model converged. If it did not some information about the reason for this is reported.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
```

```
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Select all rows without missing values

ds.completeCases(x1 = "D", newobj = "D.comp", datasources = connections)

# Fit a Poisson regression model

ds.glmerSLMA(formula = "LAB_TSC ~ LAB_HDL + (1 | GENDER)",
             offset = NULL,
             dataName = "D.comp",
             datasources = connections,
             family = "poisson")

# Clear the Datashield R sessions and logout
datashield.logout(connections)

builder <- DSI::newDSLoginBuilder()
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CLUSTER.CLUSTER_SLO1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CLUSTER.CLUSTER_SLO2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CLUSTER.CLUSTER_SLO3", driver = "OpalDriver")
logindata <- builder$build()

 # Log onto the remote Opal training servers
 connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


   # Fit a Logistic regression model

ds.glmerSLMA(formula = "Male ~  incid_rate +diabetes + (1 | age)",
             dataName = "D",
             datasources = connections[2],#only the second server is used (study2)
             family = "binomial")


# Clear the Datashield R sessions and logout
datashield.logout(connections)
```

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

`ds.glmPredict` *Applies predict.glm() to a serverside glm object*

Description

Applies native R's predict.glm() function to a serverside glm object previously created using ds.glmSLMA.

Usage

```
ds.glmPredict(
  glmname = NULL,
  newdataname = NULL,
  output.type = "response",
  se.fit = FALSE,
  dispersion = NULL,
  terms = NULL,
  na.action = "na.pass",
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`glmname`	is a character string identifying the glm object on serverside to which predict.glm is to be applied. Equivalent to <object> argument in native R's predict.glm which is described as: a fitted object of class inheriting from 'glm'.
`newdataname`	is a character string identifying an (optional) dataframe on the serverside in which to look for new covariate values with which to predict. If omitted, the original fitted linear predictors from the original glm fit are used as the basis of prediction. Precisely equivalent to the <newdata> argument in the predict.glm function in native R.
`output.type`	a character string taking the values 'response', 'link' or 'terms'. The value 'response' generates predictions on the scale of the original outcome, e.g. as proportions in a logistic regression. These are often called 'fitted values'. The value 'link' generates predictions on the scale of the linear predictor, e.g. log-odds in logistic regression, log-rate or log-count in Poisson regression. The predictions using 'response' and 'link' are identical for a standard Gaussian model with an identity link. The value 'terms' returns either fitted values or predicted values on the link scale based not on the whole linear predictor but on separate 'terms'. So, if age is modelled as a five level factor, one of the output components will relate to predictions (fitted values or link scale predictions) based on all five levels

of age simultaneously. Any simple covariate (e.g. not a composite factor) will be treated as a term in its own right. ds.glmPredict's <output.type> argument is precisely equivalent to the <type> argument in native R's predict.glm function.

`se.fit` logical if standard errors for the fitted predictions are required. Defaults to FALSE when the output contains only a vector (or vectors) of predicted values. If TRUE, the output also contains corresponding vectors for the standard errors of the predicted values, and a single value reporting the scale parameter of the model. ds.glmPredict's <se.fit> argument is precisely equivalent to the corresponding argument in predict.glm in native R. argument is equivalent to the <type> argument in native R's predict.glm function.

`dispersion` numeric value specifying the dispersion of the GLM fit to be assumed in computing the standard errors. If omitted, that returned by summary applied to the glm object is used. e.g. if <dispersion> is unspecified the dispersion assumed for a logistic regression or Poisson model is 1. But if dispersion is set to 4, the standard errors of the predictions will all be multiplied by 2 (i.e. sqrt(4)). This is useful in making predictions from models subject to overdispersion. ds.glmPredict's <dispersion> argument is precisely equivalent to the corresponding argument in predict.glm in native R.

`terms` a character vector specifying a subset of terms to return in the prediction. Only applies if output.type='terms'. ds.glmPredict's <terms> argument is precisely equivalent to the corresponding argument in predict.glm in native R.

`na.action` character string determining what should be done with missing values in the data.frame identified by <newdataname>. Default is na.pass which predicts from the specified new data.frame with all NAs left in place. na.omit removes all rows containing NAs. na.fail stops the function if there are any NAs anywhere in the data.frame. For further details see help in native R.

`newobj` a character string specifying the name of the serverside object to which the output object from the call to ds.glmPredict is to be written in each study. If no <newobj> argument is specified, the output object on the serverside defaults to the name "predict_glm".

`datasources` specifies the particular 'connection object(s)' to use. e.g. if you have several data sets in the sources you are working with called opals.a, opals.w2, and connection.xyz, you can choose which of these to work with. The call 'datashield.connections_find()' lists all of the different datasets available and if one of these is called 'default.connections' that will be the dataset used by default if no other dataset is specified. If you wish to change the connections you wish to use by default the call datashield.connections_default('opals.a') will set 'default.connections' to be 'opals.a' and so in the absence of specific instructions to the contrary (e.g. by specifying a particular dataset to be used via the <datasources> argument) all subsequent function calls will be to the datasets held in opals.a. If the <datasources> argument is specified, it should be set without inverted commas: e.g. datasources=opals.a or datasources=default.connections. The <datasources> argument also allows you to apply a function solely to a subset of the studies/sources you are working with. For example, the second source in a set of three, can be specified using a call such as datasources=connection.xyz[2]. On the other hand, if you wish to specify solely the first and third sources, the appropriate call will be datasources=connections.xyz[c(1,3)]

Details

Clientside function calling a single assign function (glmPredictDS.as) and a single aggregate function (glmPredictDS.ag). ds.glmPredict applies the native R predict.glm function to a glm object that has already been created on the serverside by fitting ds.glmSLMA. This is precisely the same as the glm object created in native R by fitting a glm using the glm function. Crucially, if ds.glmSLMA was originally applied to multiple studies the glm object created on each study is based solely on data from that study. ds.glmPredict has two distinct actions. First, the call to the assign function applies the standard predict.glm function of native R to the glm object on the serverside and writes all the output that would normally be generated by predict.glm to a newobj on the serverside. Because no critical information is passed to the clientside, there are no disclosure issues associated with this action. Any standard DataSHIELD functions can then be applied to the newobj to interpret the output. For example, it could be used as the basis for regression diagnostic plots. Second, the call to the aggregate function creates a non-disclosive summary of all the information held in the newobj created by the assign function and returns this summary to the clientside. For example, the full list of predicted/fitted values generated by the model could be disclosive. So although the newobj holds the full vector of fitted values, only the total number of values, the total number of valid (non-missing) values, the number of missing values, the mean and standard deviation of all valid values and the 5 are returned to the clientside by the aggregate function. The non-DataSHIELD arguments of ds.glmPredict are precisely the equivalent to those of predict.glm in native R and so all detailed information can be found using help(predict.glm) in native R.

Value

ds.glmPredict calls the serverside assign function glmPredictDS.as which writes a new object to the serverside containing output precisely equivalent to predict.glm in native R. The name for this serverside object is given by the newobj argument or if that argument is missing or null it is called "predict_glm". In addition, ds.glmPredict calls the serverside aggregate function glmPredictDS.ag which returns an object containing non-disclosive summary statistics relating either to a single prediction vector called fit or, if se.fit=TRUE, of two vectors 'fit' and 'se.fit' - the latter containing the standard errors of the predictions in 'fit'. The non-disclosive summary statistics for the vector(s) include: length, the total number of valid (non-missing) values, the number of missing values, the mean and standard deviation of the valid values and the 5 the output always includes: the name of the serverside glm object being predicted from, the name - if one was specified - of the dataframe being used as the basis for predictions, the output.type specified ('link', 'response' or 'terms'), the value of the dispersion parameter if one had been specified and the residual scale parameter (which is multiplied by sqrt(dispersion parameter) if one has been set). If output.type = 'terms', the summary statistics for the fit and se.fit vectors are replaced by equivalent summary statistics for each column in fit and se.fit matrices which each have k columns if k terms are being summarised.

Author(s)

Paul Burton, for DataSHIELD Development Team 13/08/20

`ds.glmSLMA` *Fit a Generalized Linear Model (GLM) with pooling via Study Level Meta-Analysis (SLMA)*

Description

Fits a generalized linear model (GLM) on data from single or multiple sources with pooled co-analysis across studies being based on SLMA (Study Level Meta Analysis).

Usage

```
ds.glmSLMA(
  formula = NULL,
  family = NULL,
  offset = NULL,
  weights = NULL,
  combine.with.metafor = TRUE,
  newobj = NULL,
  dataName = NULL,
  checks = FALSE,
  maxit = 30,
  notify.of.progress = FALSE,
  datasources = NULL
)
```

Arguments

formula	an object of class formula describing the model to be fitted. For more information see **Details**.
family	identifies the error distribution function to use in the model.
offset	a character string specifying the name of a variable to be used as an offset.ds.glmSLMA does not allow an offset vector to be written directly into the GLM formula.
weights	a character string specifying the name of a variable containing prior regression weights for the fitting process. ds.glmSLMA does not allow a weights vector to be written directly into the GLM formula.
combine.with.metafor	logical. If TRUE the estimates and standard errors for each regression coefficient are pooled across studies using random-effects meta-analysis under maximum likelihood (ML), restricted maximum likelihood (REML) or fixed-effects meta-analysis (FE). Default TRUE.
newobj	a character string specifying the name of the object to which the glm object representing the model fit on the serverside in each study is to be written. If no <newobj> argument is specified, the output object defaults to "new.glm.obj".
dataName	a character string specifying the name of an (optional) data frame that contains all of the variables in the GLM formula.
checks	logical. If TRUE ds.glmSLMA checks the structural integrity of the model. Default FALSE. For more information see **Details**.
maxit	a numeric scalar denoting the maximum number of iterations that are permitted before ds.glmSLMA declares that the model has failed to converge. For more information see **Details**.

notify.of.progress
: specifies if console output should be produced to indicate progress. Default FALSE.

datasources
: a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

ds.glmSLMA specifies the structure of a Generalized Linear Model to be fitted separately on each study or data source. Calls serverside functions glmSLMADS1 (aggregate),glmSLMADS2 (aggregate) and glmSLMADS.assign (assign). From a mathematical perspective, the SLMA approach (using ds.glmSLMA) differs fundamentally from the alternative approach using ds.glm. ds.glm fits the model iteratively across all studies together. At each iteration the model in every data source has precisely the same coefficients so when the model converges one essentially identifies the model that best fits all studies simultaneously. This mathematically equivalent to placing all individual-level data from all sources in one central warehouse and analysing those data as one combined dataset using the conventional glm() function in native R. In contrast ds.glmSLMA sends a command to every data source to fit the model required but each separate source simply fits that model to completion (ie undertakes all iterations until the model converges) and the estimates (regression coefficients) and their standard errors from each source are sent back to the client and are then pooled using SLMA via any approach the user wishes to implement. The ds.glmSLMA functions includes an argument <combine.with.metafor> which if TRUE (the default) pools the models across studies using the metafor function (from the metafor package) using three optimisation methods: random effects under maximum likelihood (ML); random effects under restricted maximum likelihood (REML); or fixed effects (FE). But once the estimates and standard errors are on the clientside, the user can alternatively choose to use the metafor package in any way he/she wishes, to pool the coefficients across studies or, indeed, to use another meta-analysis package, or their own code.

Although the ds.glm approach might at first sight appear to be preferable under all circumstances, this is not always the case. First, the results from both approaches are generally very similar. Secondly, the SLMA approach can offer key inferential advantages when there is marked heterogeneity between sources that cannot simply be corrected by including fixed-effects in one's ds.glm model that each reflect a study- or centre-specific effect. In particular, such fixed effects cannot be guaranteed to generate formal inferences that are unbiased when there is heterogeneity in the effect that is actually of scientific interest. It might be argued that one should not try to pool the inferences anyway if there is marked heterogeneity, but you can use the joint analysis to formally check for such heterogeneity and then choose to report the pooled result or separate results from each study individually. Crucially, unless the heterogeneity is substantial, pooling can be quite reasonable. Furthermore, if you just fit a ds.glm model without centre-effects you will in effect be pooling across all studies without checking for heterogeneity and if heterogeneity exists and if it is strong you can get theoretically results that are badly confounded by study. Before we introduced ds.glmSLMA we encountered a real world example of a ds.glm (without centre effects) which generated combined inferences over all studies which were more extreme than the results from any of the individual studies: the lower 95 of the combined estimate was higher than the upper 95 ALL of the individual studies. This was clearly incorrect and provided a salutary lesson on the potential impact of confounding by study if a ds.glm model does not include appropriate centre-effects. Even if you are going to undertake a ds.glm analysis (which is slightly more powerful when there is no heterogeneity) it may still be useful to also carry out a ds.glmSLMA analysis as this provides a very easy way to examine the extent of heterogeneity.

In `formula` Most shortcut notation for formulas allowed under R's standard `glm()` function is also allowed by `ds.glmSLMA`.

Many glms can be fitted very simply using a formula such as:

$y\ a + b + c + d$

which simply means fit a glm with `y` as the outcome variable and `a`, `b`, `c` and `d` as covariates. By default all such models also include an intercept (regression constant) term.

Instead, if you need to fit a more complex model, for example:

$EVENT\ 1 + TID + SEXF * AGE.60$

In the above model the outcome variable is `EVENT` and the covariates `TID` (factor variable with level values between 1 and 6 denoting the period time), `SEXF` (factor variable denoting sex) and `AGE.60` (quantitative variable representing age-60 in years). The term `1` forces the model to include an intercept term, in contrast if you use the term `0` the intercept term is removed. The `*` symbol between `SEXF` and `AGE.60` means fit all possible main effects and interactions for and between those two covariates. This takes the value 0 in all males `0 * AGE.60` and in females `1 * AGE.60`. This model is in example 1 of the section **Examples**. In this case the logarithm of the survival time is added as an offset (`log(survtime)`).

In the `family` argument a range of model types can be fitted. This range has recently been extended to include a number of model types that are non-standard but are used relatively widely.

The standard models include:

`"gaussian"` : conventional linear model with normally distributed errors

`"binomial"` : conventional unconditional logistic regression model

`"poisson"` : Poisson regression model which is often used in epidemiological analysis of counts and rates and is also used in survival analysis. The Piecewise Exponential Regression (PER) model typically provides a close approximation to the Cox regression model in its main estimates and standard errors.

`"gamma"` : a family of models for outcomes characterised by a constant coefficient of variation, i.e. the variance increases with the square of the expected mean

The extended range includes:

`"quasipoisson"` : a model with a Poisson variance function - variance equals expected mean - but the residual variance which is fixed to be 1.00 in a standard Poisson model can then take any value. This is achieved by a dispersion parameter which is estimated during the model fit and if it takes the value K it means that the expected variance is K x the expected mean, which implies that all standard errors will be sqrt(K) times larger than in a standard Poisson model fitted to the same data. This allows for the extra uncertainty which is associated with 'overdispersion' that occurs very commonly with Poisson distributed data, and typically arises when the count/rate data being modelled occur in blocks which exhibit heterogeneity of underlying risk which is not being fully modelled, either by including the blocks themselves as a factor or by including covariates for all the determinants that are relevant to that underlying risk. If there is no overdispersion (K=1) the estimates and standard errors from the quasipoisson model will be almost identical to those from a standard poisson model.

`"quasibinomial"` : a model with a binomial variance function - if P is the expected proportion of successes, and N is the number of "trials" (always 1 if analysing binary data which are formally described as having a Bernoulli distribution (binomial distribution with N=1) the variance function is N*(P)*(1-P). But the residual variance which is fixed to be 1.00 in a

binomial model can take any value. This is achieved by a dispersion parameter which is estimated during the model fit (see quasipoisson information above).

Each class of models has a "canonical link" which represents the link function that maximises the information extraction by the model. The gaussian family uses the `identity` link, the poisson family the `log` link, the binomial/Bernoulli family the `logit` link and the the gamma family the `reciprocal` link.

The `dataName` argument avoids you having to specify the name of the data frame in front of each covariate in the formula. For example, if the data frame is called `DataFrame` you avoid having to write: $DataFrame\$y\ DataFrame\$a + DataFrame\$b + DataFrame\$c + DataFrame\$d$

The `checks` argument verifies that the variables in the model are all defined (exist) on the server-site at every study and that they have the correct characteristics required to fit the model. It is suggested to make `checks` argument TRUE only if an unexplained problem in the model fit is encountered because the running process takes several minutes.

In `maxit` Logistic regression and Poisson regression models can require many iterations, particularly if the starting value of the regression constant is far away from its actual value that the GLM is trying to estimate. In consequence we often set `maxit=30` but depending on the nature of the models you wish to fit, you may wish to be alerted much more quickly than this if there is a delay in convergence, or you may wish to allow more iterations.

Server functions called: `glmSLMADS1`, `glmSLMADS2`, `glmSLMADS.assign`

Value

The serverside aggregate functions `glmSLMADS1` and `glmSLMADS2` return output to the clientside, while the assign function `glmSLMADS.assign` simply writes the glm object to the serverside created by the model fit on a given server as a permanent object on that same server. This is precisely the same as the glm object that is usually created by a call to glm() in native R and it contains all the same elements (see help for glm in native R). Because it is a serverside object, no disclosure blocks apply. However, such disclosure blocks do apply to the information passed to the clientside. In consequence, rather than containing all the components of a standard glm object in native R, the components of the glm object that are returned by `ds.glmSLMA` include: a mixture of non-disclosive elements of the glm object reported separately by study included in a list object called `output.summary`; and a series of other list objects that represent inferences aggregated across studies.

the study specific items include:

`coefficients`: a matrix with 5 columns:

First : the names of all of the regression parameters (coefficients) in the model

second : the estimated values

third : corresponding standard errors of the estimated values

fourth : the ratio of estimate/standard error

fifth : the p-value treating that as a standardised normal deviate

`family`: indicates the error distribution and link function used in the GLM.

`formula`: model formula, see description of formula as an input parameter (above).

`df.resid`: the residual degrees of freedom around the model.

deviance.resid: the residual deviance around the model.

df.null: the degrees of freedom around the null model (with just an intercept).

dev.null: the deviance around the null model (with just an intercept).

CorrMatrix: the correlation matrix of parameter estimates.

VarCovMatrix: the variance-covariance matrix of parameter estimates.

weights: the name of the vector (if any) holding regression weights.

offset: the name of the vector (if any) holding an offset (enters glm with a coefficient of 1.00).

cov.scaled: equivalent to VarCovMatrix.

cov.unscaled: equivalent to VarCovMatrix but assuming dispersion (scale) parameter is 1.

Nmissing: the number of missing observations in the given study.

Nvalid: the number of valid (non-missing) observations in the given study.

Ntotal: the total number of observations in the given study (Nvalid + Nmissing).

data: equivalent to input parameter dataName (above).

dispersion: the estimated dispersion parameter: deviance.resid/df.resid for a gaussian family multiple regression model, 1.00 for logistic and poisson regression.

call: summary of key elements of the call to fit the model.

na.action: chosen method of dealing with missing values. This is usually, na.action = na.omit - see help in native R.

iter: the number of iterations required to achieve convergence of the glm model in each separate study.

Once the study-specific output has been returned, ds.glmSLMA returns a series of lists relating to the aggregated inferences across studies. These include the following:

num.valid.studies: the number of studies with valid output included in the combined analysis

betamatrix.all: matrix with a row for each regression coefficient and a column for each study reporting the estimated regression coefficients by study.

sematrix.all: matrix with a row for each regression coefficient and a column for each study reporting the standard errors of the estimated regression coefficients by study.

betamatrix.valid: matrix with a row for each regression coefficient and a column for each study reporting the estimated regression coefficients by study but only for studies with valid output (eg not violating disclosure traps)

sematrix.all: matrix with a row for each regression coefficient and a column for each study reporting the standard errors of the estimated regression coefficients by study but only for studies with valid output (eg not violating disclosure traps)

SLMA.pooled.estimates.matrix: a matrix with a row for each regression coefficient and six columns. The first two columns contain the pooled estimate of each regression coefficients and its standard error with pooling via random effect meta-analysis under maximum likelihood (ML). Columns 3 and 4 contain the estimates and standard errors from random effect meta-analysis under REML and columns 5 and 6 the estimates and standard errors under fixed effect meta-analysis. This matrix is only returned if the argument combine.with.metafor is set to TRUE. Otherwise, users can take the betamatrix.valid and sematrix.valid matrices and enter them into their meta-analysis package of choice.

`is.object.created` and `validity.check` are standard items returned by an assign function when the designated newobj appears to have been successfully created on the serverside at each study. This output is produced specifically by the assign function `glmSLMADS.assign` that writes out the glm object on the serverside

Author(s)

Paul Burton, for DataSHIELD Development Team 07/07/20

Examples

```
## Not run:

 ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  # Example 1: Fitting GLM for survival analysis
  # For this analysis we need to load survival data from the server

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Fit the GLM

  # make sure that the outcome is numeric
  ds.asNumeric(x.name = "D$cens",
               newobj = "EVENT",
               datasources = connections)

  # convert time id variable to a factor

  ds.asFactor(input.var.name = "D$time.id",
              newobj = "TID",
```

```
            datasources = connections)

  # create in the server-side the log(survtime) variable

  ds.log(x = "D$survtime",
         newobj = "log.surv",
         datasources = connections)

  ds.glmSLMA(formula = EVENT ~ 1 + TID + female * age.60,
         dataName = "D",
         family = "poisson",
         offset = "log.surv",
         weights = NULL,
         checks = FALSE,
         maxit = 20,
         datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

  # Example 2: run a logistic regression without interaction
  # For this example we are going to load another type of data

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Fit the logistic regression model

  mod <- ds.glmSLMA(formula = "DIS_DIAB~GENDER+PM_BMI_CONTINUOUS+LAB_HDL",
                dataName = "D",
                family = "binomial",
                datasources = connections)

  mod #visualize the results of the model

# Example 3: fit a standard Gaussian linear model with an interaction
# We are using the same data as in example 2. It is not necessary to
# connect again to the server
```

```
mod <- ds.glmSLMA(formula = "PM_BMI_CONTINUOUS~DIS_DIAB*GENDER+LAB_HDL",
              dataName = "D",
              family = "gaussian",
              datasources = connections)
mod

# Clear the Datashield R sessions and logout
datashield.logout(connections)

## End(Not run)
```

---

ds.glmSummary *Summarize a glm object on the serverside*

Description

Summarize a glm object on the serverside to create a summary_glm object. Also identify and return components of both the glm object and the summary_glm object that can safely be sent to the clientside without a risk of disclosure

Usage

```
ds.glmSummary(x.name, newobj = NULL, datasources = NULL)
```

Arguments

x.name	a character string providing the name of a glm object on the serverside that has previously been created e.g. using ds.glmSLMA
newobj	a character string specifying the name of the object to which the summary_glm object representing the output of summary(glm object) in each study is to be written. If no <newobj> argument is specified, the output object on the serverside defaults to "summary_glm.newobj".
datasources	specifies the particular 'connection object(s)' to use. e.g. if you have several data sets in the sources you are working with called opals.a, opals.w2, and connection.xyz, you can choose which of these to work with. The call 'datashield.connections_find()' lists all of the different datasets available and if one of these is called 'default.connections' that will be the dataset used by default if no other dataset is specified. If you wish to change the connections you wish to use by default the call datashield.connections_default('opals.a') will set 'default.connections' to be 'opals.a' and so in the absence of specific instructions to the contrary (e.g. by specifying a particular dataset to be used via the <datasources> argument) all subsequent function calls will be to the datasets held in opals.a. If the <datasources> argument is specified, it should be set without inverted commas: e.g. datasources=opals.a or datasources=default.connections. The <datasources> argument also allows you to apply a function solely to a subset of the studies/sources you are working with. For example, the second source

in a set of three, can be specified using a call such as datasources=connection.xyz[2]. On the other hand, if you wish to specify solely the first and third sources, the appropriate call will be datasources=connections.xyz[c(1,3)]

Details

Clientside function calling a single assign function (glmSummaryDS.as) and a single aggregate function (glmSummaryDS.as). ds.glmSummary summarises a glm object that has already been created on the serverside by fitting ds.glmSLMA which is precisely the same as the glm object created by fitting a glm using the glm function in native R. Similarly the summary_glm object saved to the serverside is precisely equivalent to the object created using summary(glm object) in R. The glm object produced from a standard call to glm in R has 32 components. Amongst these, all of the following thirteen contain information about every records in the data set and so are disclosive. They are all therefore set to NA and so convey no information when returned to the clientside: 1.residuals, 2.fitted.values, 3.effects, 4.R, 5.qr, 6.linear.predictors, 7.weights, 8.prior.weights, 9.y, 10.model, 11. na.action, 12.x, 13. offset. In addition the list element "data" which identifies a data.frame that was identified as containing all of the variables required for the model is also disclosive because it doesn't list the name of the data.frame but rather prints it out in full. However, a user can benefit from knowing what source of data were used in creating the glm model and so the element "data" that is returned to the clientside simply lists the names of all of the columns in the originating data.frame. Having removed all disclosive elements of the glm object, ds.glmSummary returns the remaining 19 elements to the clientside. The object created from a standard call to summary(glm object) in R contains 18 list elements. Only two of these are disclosive - na.action and deviance.resid and these are therefore set to NA before ds.glmSummary returns the other 16 to the clientside. Further details of the components of the glm object and summary_glm object can be found under help for glm and summary(glm) in native R. In addition, the elements that ARE returned are listed under "return" below.

Value

ds.glmSummary writes a new object to the serverside with name given by the newobj argument or if that argument is missing or null it is called "summary_glm.newobj". In addition, ds.glmSummary returns an object containing two lists to the clientside the two lists are named "glm.obj" and "glm.summary.obj" which contain all of the elements of the original glm object and the summary_glm object on the serverside but with all potentially disclosive components set to NA or masked in another way see "details" above. The elements that are returned with a non-NA value in the glm.obj list object are: "coefficients", "rank", "family", "deviance", "aic", "null.deviance", "iter", "df.residual", "df.null", "converged", "boundary", "call", "formula", "terms", "data", "control", "method", "contrasts", "xlevels". The elements that are returned with a non-NA value in the glm.summary.obj list object are: "call", "terms", "family", "deviance", "aic", "contrasts", "df.residual", "null.deviance", "df.null", "iter", "coefficients", "aliased", "dispersion", "df", "cov.unscaled", "cov.scaled". For further information see help for glm and summary(glm) in native R and for ds.glmSLMA in DataSHIELD.

Author(s)

Paul Burton, for DataSHIELD Development Team 17/07/20

`ds.heatmapPlot` *Generates a Heat Map plot*

Description

Generates a heat map plot of the pooled data or one plot for each dataset.

Usage

```
ds.heatmapPlot(
  x = NULL,
  y = NULL,
  type = "combine",
  show = "all",
  numints = 20,
  method = "smallCellsRule",
  k = 3,
  noise = 0.25,
  datasources = NULL
)
```

Arguments

`x`	a character string specifying the name of a numerical vector.
`y`	a character string specifying the name of a numerical vector.
`type`	a character string that represents the type of graph to display. `type` argument can be set as `'combine'` or `'split'`. Default `'combine'`. For more information see **Details**.
`show`	a character string that represents where the plot should be focused. `show` argument can be set as `'all'` or `'zoomed'`. Default `'all'`. For more information see **Details**.
`numints`	the number of intervals for a density grid object. Default `numints` value is `20`.
`method`	a character string that defines which heat map will be created. The `method` argument can be set as `'smallCellsRule'`, `'deterministic'` or `'probabilistic'`. Default `'smallCellsRule'`. For more information see **Details**.
`k`	the number of the nearest neighbours for which their centroid is calculated. Default `k` value is `3`. For more information see **Details**.
`noise`	the percentage of the initial variance that is used as the variance of the embedded noise if the argument `method` is set to `'probabilistic'`. Default `noise` value is `0.25`. For more information see **Details**.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The ds.heatmapPlot function first generates a density grid and uses it to plot the graph. Cells of the grid density matrix that hold a count of less than the filter set by DataSHIELD (usually 5) are considered invalid and turned into 0 to avoid potential disclosure. A message is printed to inform the user about the number of invalid cells. The ranges returned by each study and used in the process of getting the grid density matrix are not the exact minimum and maximum values but rather close approximates of the real minimum and maximum value. This was done to reduce the risk of potential disclosure.

In the argument type can be specified two types of graphics to display:

'combine' : a combined heat map plot is displayed

'split' : each heat map is plotted separately

In the argument show can be specified two options:

'all' : the ranges of the variables are used as plot limits

'zoomed' : the plot is zoomed to the region where the actual data are

In the argument method can be specified 3 different heat map to be created:

'smallCellsRule' : the heat map of the actual variables is created but grids with low counts are replaced with grids with zero counts

'deterministic' : the heat map of the scaled centroids of each k nearest neighbours of the original variables are created, where the value of k is set by the user

'probabilistic' : the heat map of 'noisy' variables is generated. The added noise follows a normal distribution with zero mean and variance equal to a percentage of the initial variance of each input variable. This percentage is specified by the user in the argument noise

In the k argument the user can choose any value for k equal to or greater than the pre-specified threshold used as a disclosure control for this method and lower than the number of observations minus the value of this threshold. By default the value of k is set to be equal to 3 (we suggest k to be equal to, or bigger than, 3). Note that the function fails if the user uses the default value but the study has set a bigger threshold. The value of k is used only if the argument method is set to 'deterministic'. Any value of k is ignored if the argument method is set to 'probabilistic' or 'smallCellsRule'.

The value of noise is used only if the argument method is set to 'probabilistic'. Any value of noise is ignored if the argument method is set to 'deterministic' or 'smallCellsRule'. The user can choose any value for noise equal to or greater than the pre-specified threshold 'nfilter.noise'.

Server function called: heatmapPlotDS

Value

ds.heatmapPlot returns to the client-side a heat map plot and a message specifying the number of invalid cells in each study.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Compute the heat map plot
  # Example 1: Plot a combined (default) heat map plot of the variables 'LAB_TSC'
  # and 'LAB_HDL' using the method 'smallCellsRule' (default)
  ds.heatmapPlot(x = 'D$LAB_TSC',
                 y = 'D$LAB_HDL',
                 datasources = connections) #all servers are used

  # Example 2: Plot a split heat map  plot of the variables 'LAB_TSC'
  # and 'LAB_HDL' using the method 'smallCellsRule' (default)
  ds.heatmapPlot(x = 'D$LAB_TSC',
                 y = 'D$LAB_HDL',
                 method = 'smallCellsRule',
                 type = 'split',
                 datasources = connections[1]) #only the first server is used (study1)

 # Example 3: Plot a combined heat map plot using the method 'deterministic' centroids of each
  # k = 7 nearest neighbours for numints = 40
  ds.heatmapPlot(x = 'D$LAB_TSC',
                 y = 'D$LAB_HDL',
                 numints = 40,
                 method = 'deterministic',
                 k = 7,
                 type = 'split',
                 datasources = connections[2]) #only the second server is used (study2)
```

```
  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.hetcor *Heterogeneous Correlation Matrix*

Description

This function is based on the hetcor function from the R package polycor.

Usage

```
ds.hetcor(
  data = NULL,
  ML = TRUE,
  std.err = TRUE,
  bins = 4,
  pd = TRUE,
  use = "complete.obs",
  datasources = NULL
)
```

Arguments

data	the name of a data frame consisting of factors, ordered factors, logical variables, character variables, and/or numeric variables, or the first of several variables.
ML	if TRUE, compute maximum-likelihood estimates; if FALSE (default), compute quick two-step estimates.
std.err	if TRUE (default), compute standard errors.
bins	number of bins to use for continuous variables in testing bivariate normality; the default is 4.
pd	if TRUE (default) and if the correlation matrix is not positive-definite, an attempt will be made to adjust it to a positive-definite matrix, using the nearPD function in the Matrix package. Note that default arguments to nearPD are used (except corr=TRUE); for more control call nearPD directly.
use	if "complete.obs", remove observations with any missing data; if "pairwise.complete.obs", compute each correlation using all observations with valid data for that pair of variables.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Computes a heterogenous correlation matrix, consisting of Pearson product-moment correlations between numeric variables, polyserial correlations between numeric and ordinal variables, and polychoric correlations between ordinal variables.

Value

Returns an object of class "hetcor" from each study, with the following components: the correlation matrix; the type of each correlation: "Pearson", "Polychoric", or "Polyserial"; the standard errors of the correlations, if requested; the number (or numbers) of observations on which the correlations are based; p-values for tests of bivariate normality for each pair of variables; the method by which any missing data were handled: "complete.obs" or "pairwise.complete.obs"; TRUE for ML estimates, FALSE for two-step estimates.

Author(s)

Demetris Avraam for DataSHIELD Development Team

---

ds.histogram *Generates a histogram plot*

Description

`ds.histogram` function plots a non-disclosive histogram in the client-side.

Usage

```
ds.histogram(
  x = NULL,
  type = "split",
  num.breaks = 10,
  method = "smallCellsRule",
  k = 3,
  noise = 0.25,
  vertical.axis = "Frequency",
  datasources = NULL
)
```

Arguments

`x`	a character string specifying the name of a numerical vector.
`type`	a character string that represents the type of graph to display. The `type` argument can be set as `'combine'` or `'split'`. Default `'split'`. For more information see **Details**.
`num.breaks`	a numeric specifying the number of breaks of the histogram. Default value is `10`.

method	a character string that defines which histogram will be created. The method argument can be set as 'smallCellsRule', 'deterministic' or 'probabilistic'. Default 'smallCellsRule'. For more information see **Details**.
k	the number of the nearest neighbours for which their centroid is calculated. Default k value is 3. For more information see **Details**.
noise	the percentage of the initial variance that is used as the variance of the embedded noise if the argument method is set to 'probabilistic'. Default noise value is 0.25. For more information see **Details**.
vertical.axis	a character string that defines what is shown in the vertical axis of the plot. The vertical.axis argument can be set as 'Frequency' or 'Density'. Default 'Frequency'. For more information see **Details**.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

ds.histogram function allows the user to plot distinct histograms (one for each study) or a combined histogram that merges the single plots.

In the argument type can be specified two types of graphics to display:

'combine' : a histogram that merges the single plot is displayed.

'split' : each histogram is plotted separately.

In the argument method can be specified 3 different histograms to be created:

'smallCellsRule' : the histogram of the actual variable is created but bins with low counts are removed.

'deterministic' : the histogram of the scaled centroids of each k nearest neighbours of the original variable where the value of k is set by the user.

'probabilistic' : the histogram shows the original distribution disturbed by the addition of random stochastic noise. The added noise follows a normal distribution with zero mean and variance equal to a percentage of the initial variance of the input variable. This percentage is specified by the user in the argument noise.

In the k argument the user can choose any value for k equal to or greater than the pre-specified threshold used as a disclosure control for this method and lower than the number of observations minus the value of this threshold. By default the value of k is set to be equal to 3 (we suggest k to be equal to, or bigger than, 3). Note that the function fails if the user uses the default value but the study has set a bigger threshold. The value of k is used only if the argument method is set to 'deterministic'. Any value of k is ignored if the argument method is set to 'probabilistic' or 'smallCellsRule'.

In the noise argument the percentage of the initial variance that is used as the variance of the embedded noise if the argument method is set to 'probabilistic'. Any value of noise is ignored if the argument method is set to 'deterministic' or 'smallCellsRule'. The user can choose any value for noise equal to or greater than the pre-specified threshold 'nfilter.noise'. By default the value of noise is set to be equal to 0.25.

In the argument vertical.axis can be specified two types of histograms:

'Frequency' : the histogram of the frequencies is returned.

'Density' : the histogram of the densities is returned.

Server function called: histogramDS2

Value

one or more histogram objects and plots depending on the argument type

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Compute the histogram
  # Example 1: generate a histogram for each study separately
  ds.histogram(x = 'D$PM_BMI_CONTINUOUS',
               type = "split",
               datasources = connections) #all studies are used

  # Example 2: generate a combined histogram with the default small cells counts
  #             suppression rule
  ds.histogram(x = 'D$PM_BMI_CONTINUOUS',
                method = 'smallCellsRule',
```

```
              type = 'combine',
              datasources = connections[1]) #only the first study is used (study1)

  # Example 3: if a variable is of type factor the function returns an error
  ds.histogram(x = 'D$PM_BMI_CATEGORICAL',
              datasources = connections)

  # Example 4: generate a combined histogram with the deterministic method for k=50
  ds.histogram(x = 'D$PM_BMI_CONTINUOUS',
              k = 50,
              method = 'deterministic',
              type = 'combine',
              datasources = connections[2])#only the second study is used (study2)


  # Example 5: create a histogram and the probability density on the plot
  hist <- ds.histogram(x = 'D$PM_BMI_CONTINUOUS',
                       method = 'probabilistic', type='combine',
                       num.breaks = 30,
                       vertical.axis = 'Density',
                       datasources = connections)
  lines(hist$mids, hist$density)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.igb_standards *Converts birth measurements to intergrowth z-scores/centiles*

Description

Converts birth measurements to INTERGROWTH z-scores/centiles (generic)

Usage

```
ds.igb_standards(
  gagebrth = NULL,
  z = 0,
  p = 50,
  val = NULL,
  var = NULL,
  sex = NULL,
  fun = "igb_value2zscore",
  newobj = NULL,
  datasources = NULL
)
```

Arguments

gagebrth	the name of the "gestational age at birth in days" variable.
z	z-score(s) to convert (must be between 0 and 1). Default value is 0. This value is used only if fun is set to "igb_zscore2value".
p	centile(s) to convert (must be between 0 and 100). Default value is p=50. This value is used only if fun is set to "igb_centile2value".
val	the name of the anthropometric variable to convert.
var	the name of the measurement to convert ("lencm", "wtkg", "hcircm", "wlr").
sex	the name of the sex factor variable. The variable should be coded as Male/Female. If it is coded differently (e.g. 0/1), then you can use the ds.recodeValues function to recode the categories to Male/Female before the use of ds.igb_standards.
fun	the name of the function to be used. This can be one of: "igb_centile2value", "igb_zscore2value", "igb_value2zscore" (default), "igb_value2centile".
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default name is set to igb.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Value

assigns the converted measurement as a new object on the server-side

Note

For gestational ages between 24 and 33 weeks, the INTERGROWTH very early preterm standard is used.

Author(s)

Demetris Avraam for DataSHIELD Development Team

References

- Villar, J., Ismail, L.C., Victora, C.G., Ohuma, E.O., Bertino, E., Altman, D.G., Lambert, A., Papageorghiou, A.T., Carvalho, M., Jaffer, Y.A., Gravett, M.G., Purwar, M., Frederick, I.O., Noble, A.J., Pang, R., Barros, F.C., Chumlea, C., Bhutta, Z.A., Kennedy, S.H., 2014. International standards for newborn weight, length, and head circumference by gestational age and sex: the Newborn Cross-Sectional Study of the INTERGROWTH-21st Project. The Lancet 384, 857–868. https://doi.org/10.1016/S0140-6736(14)60932-6
- Villar, J., Giuliani, F., Fenton, T.R., Ohuma, E.O., Ismail, L.C., Kennedy, S.H., 2016. INTERGROWTH-21st very preterm size at birth reference charts. The Lancet 387, 844–845. https://doi.org/10.1016/S0140-6736(16)00384-6

ds.isNA *Checks if a server-side vector is empty*

Description

this function is similar to R function is.na but instead of a vector of booleans it returns just one boolean to tell if all the elements are missing values.

Usage

```
ds.isNA(x = NULL, datasources = NULL)
```

Arguments

x	a character string specifying the name of the vector to check.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

In certain analyses such as GLM none of the variables should be missing at complete (i.e. missing value for each observation). Since in DataSHIELD it is not possible to see the data it is important to know whether or not a vector is empty to proceed accordingly.

Server function called: isNaDS

Value

ds.isNA returns a boolean. If it is TRUE the vector is empty (all values are NA), FALSE otherwise.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
```

```
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# check if all the observation of the variable 'LAB_HDL' are missing (NA)
ds.isNA(x = 'D$LAB_HDL',
        datasources = connections) #all servers are used
ds.isNA(x = 'D$LAB_HDL',
        datasources = connections[1]) #only the first server is used (study1)


# clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

`ds.isValid` *Checks if a server-side object is valid*

Description

Checks if a vector or table structure has a number of observations equal to or greater than the threshold set by DataSHIELD.

Usage

```
ds.isValid(x = NULL, datasources = NULL)
```

Arguments

`x`	a character string specifying the name of a vector, dataframe or matrix.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

In DataSHIELD, analyses are possible only on valid objects to ensure the output is not disclosive. This function checks if an input object is valid. A vector is valid if the number of observations is equal to or greater than a set threshold. A factor vector is valid if all its levels (categories) have a count equal or greater than the set threshold. A data frame or a matrix is valid if the number of rows is equal or greater than the set threshold.

Server function called: `isValidDS`

Value

`ds.isValid` returns a boolean. If it is TRUE input object is valid, FALSE otherwise.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Check if the dataframe assigned above is valid
  ds.isValid(x = 'D',
             datasources = connections) #all servers are used
  ds.isValid(x = 'D',
             datasources = connections[2]) #only the second server is used (study2)
```

```
  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.kurtosis *Calculates the kurtosis of a numeric variable*

Description

This function calculates the kurtosis of a numeric variable.

Usage

```
ds.kurtosis(x = NULL, method = 1, type = "both", datasources = NULL)
```

Arguments

x	a string character, the name of a numeric variable.
method	an integer between 1 and 3 selecting one of the algorithms for computing kurtosis detailed below. The default value is set to 1.
type	a character which represents the type of analysis to carry out. If type is set to 'combine', 'combined', 'combines' or 'c', the global kurtosis is returned if type is set to 'split', 'splits' or 's', the kurtosis is returned separately for each study. if type is set to 'both' or 'b', both sets of outputs are produced. The default value is set to 'both'.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The function calculates the kurtosis of an input variable x with three different methods. The method is specified by the argument method. If x contains any missings, the function removes those before the calculation of the kurtosis. If method is set to 1 the following formula is used $kurtosis = \frac{\sum_{i=1}^{N}(x_i-\bar{(x)})^4/N}{(\sum_{i=1}^{N}((x_i-\bar{(x)})^2)/N)^{(2)}} - 3$, where $\bar{x}$ is the mean of x and N is the number of observations. If method is set to 2 the following formula is used $kurtosis = ((N+1) * (\frac{\sum_{i=1}^{N}(x_i-\bar{(x)})^4/N}{(\sum_{i=1}^{N}((x_i-\bar{(x)})^2)/N)^{(2)}} - 3) + 6) * ((N-1)/((N-2) * (N-3)))$. If method is set to 3 the following formula is used $kurtosis = (\frac{\sum_{i=1}^{N}(x_i-\bar{(x)})^4/N}{(\sum_{i=1}^{N}((x_i-\bar{(x)})^2)/N)^{(2)}}) * (1-1/N)^2 - 3$. This function is similar to the function kurtosis in R package e1071.

Value

a matrix showing the kurtosis of the input numeric variable, the number of valid observations and the validity message.

Author(s)

Demetris Avraam, for DataSHIELD Development Team

`ds.length` *Gets the length of an object in the server-side*

Description

This function gets the length of a vector or list that is stored on the server-side. This function is similar to the R function `length`.

Usage

```
ds.length(x = NULL, type = "both", checks = "FALSE", datasources = NULL)
```

Arguments

`x`	a character string specifying the name of a vector or list.
`type`	a character that represents the type of analysis to carry out. If `type` is set to `'combine'`, `'combined'`, `'combines'` or `'c'`, a global length is returned if `type` is set to `'split'`, `'splits'` or `'s'`, the length is returned separately for each study. if `type` is set to `'both'` or `'b'`, both sets of outputs are produced. Default `'both'`.
`checks`	logical. If TRUE the model components are checked. Default FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Server function called: `lengthDS`

Value

`ds.length` returns to the client-side the pooled length of a vector or a list, or the length of a vector or a list for each study separately.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Get the total number of observations of the vector of
  # variable 'LAB_TSC' across all the studies
  ds.length(x = 'D$LAB_TSC',
            type = 'combine',
            datasources = connections)

  # Example 2: Get the number of observations of the vector of variable
  # 'LAB_TSC' for each study separately
  ds.length(x = 'D$LAB_TSC',
            type = 'split',
            datasources = connections)

  # Example 3: Get the number of observations on each study and the total
  # number of observations across all the studies for the variable 'LAB_TSC'
  ds.length(x = 'D$LAB_TSC',
            type = 'both',
            datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

ds.levels *Produces levels attributes of a server-side factor*

Description

This function provides access to the level attribute of a factor variable stored on the server-side. This function is similar to R function levels.

Usage

```
ds.levels(x = NULL, datasources = NULL)
```

Arguments

x	a character string specifying the name of a factor variable.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Server function called: levelsDS

Value

ds.levels returns to the client-side the levels of a factor class variable stored in the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Get the levels of the PM_BMI_CATEGORICAL variable
  ds.levels(x = 'D$PM_BMI_CATEGORICAL',
            datasources = connections)#all servers are used
  ds.levels(x = 'D$PM_BMI_CATEGORICAL',
            datasources = connections[2])#only the second server is used (study2)

  # Example 2: Get the levels of the LAB_TSC variable
  # This example should not work because LAB_TSC is a continuous variable
  ds.levels(x = 'D$LAB_TSC',
            datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.lexis` *Represents follow-up in multiple states on multiple time scales*

Description

This function takes a data frame containing survival data and expands it by converting records at the level of individual subjects (survival time, censoring status, IDs and other variables) into multiple records over a series of pre-defined time intervals.

Usage

```
ds.lexis(
  data = NULL,
  intervalWidth = NULL,
  idCol = NULL,
  entryCol = NULL,
  exitCol = NULL,
  statusCol = NULL,
  variables = NULL,
  expandDF = NULL,
  datasources = NULL
)
```

Arguments

data	a character string specifying the name of a data frame containing the survival data to be expanded.
intervalWidth	a numeric vector specifying the length of each interval. For more information see **Details**.
idCol	a character string denoting the column name that holds the individual IDs of the subjects. For more information see **Details**.
entryCol	a character string denoting the column name that holds the entry times (i.e. start of follow up). For more information see **Details**.
exitCol	a character string denoting the column name that holds the exit times (i.e. end of follow up). For more information see **Details**.
statusCol	a character string denoting the column name that holds the failure/censoring status of each subject. For more information see **Details**.
variables	a vector of character strings denoting the column names of additional variables to include in the final expanded table. For more information see **Details**.
expandDF	a character string denoting the name of the new data frame containing the expanded data set. Default `lexis.newobj`.
datasources	a list of DSConnection-class objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The function `ds.lexis` splits the survival interval time of subjects into pre-specified sub-intervals that are each assumed to encompass a constant base-line hazard which means a constant instantaneous risk of death). In the expanded dataset a row is included for every interval in which a given individual is followed - regardless of how short or long that period may be. Each row includes:
(1) **CENSOR**: a variable indicating failure status for a particular interval in that interval also known as censoring status. This variable can take two values: **1** representing that the patient has died, relapsed or developed a disease. **0** representing the lost-to-follow-up or passed right through the interval without failing.
(2) **SURVTIME** an exposure-time variable indicating the duration of exposure-to-risk-of-failure the corresponding individual experienced in that interval before he/she failed or was censored.

To illustrate, an individual who survives through 5 such intervals and then dies/fails in the 6th interval will be allocated a 0 value for the failure status/censoring variable in the first five intervals and a 1 value in the 6th, while the exposure-time variable will be equal to the total length of the relevant interval in each of the first five intervals, and the additional length of time they survived in the sixth interval before they failed or were censored. If they survive through the first interval and they are censored in the second interval, the failure-status variable will take the value 0 in both intervals.
(3) **UID.expanded** the expanded data set also includes a unique ID in a form such as 77.13 which identifies that row of the dataset as relating to the 77th individual in the input data set and his/her experience (exposure-time and failure status)in the 14th interval. Note that `.N` indicates the `(N+1)`th interval because interval 1 has no suffix.
(4) **IDSEQ** the first part of `UID.expanded` (before the `'.'`). The value of this variable is repeated in every row to which the corresponding individual contributes data (i.e. to every row corresponding

to an interval in which that individual was followed).
(5) The expanded dataset contains any other variables about each individual that the user would like to carry forward to a survival analysis based on the expanded data. Typically, this will include the original ID as specified to the data repository, the total survival time (equivalent to the sum of the exposure times across all intervals) and the ultimate failure-status in the final interval to which they were exposed. The value of each of these variables is also repeated in every row corresponding to an interval in which that individual was followed.

In `intervalWidth` argument if the total sum of the duration across all intervals is less than the maximum follow-up of any individual in any contributing study, a final interval will be added by `ds.lexis` extending from the end of the last interval specified to the maximum follow-up time. If a single numeric value is specified rather than a vector, `ds.lexis` will keep adding intervals of the length specified until the maximum follow-up time in any single study is exceeded. This argument is subject to disclosure checks.

The `idCol` argument must be a numeric or character. Note that when a particular variable is identified as being the main ID to the data repository when the data are first transferred to the data repository (i.e. before DataSHIELD is used), that ID often ends up being of class character and will then be sorted in alphabetic order (treating each digit as a character) rather than numeric. For example, containing the sequential IDs 1-1000, the order of the IDs will be:
1,10,100,101,102,103,104,105,106,107,108,109,11 ...
In an alphabetic listing: NOT to the expected order:
1,2,3,4,5,6,7,8,9,10,11,12,13 ...

This alphabetic order or the ID listing will then carry forward to the expanded dataset. But the nature and order of the original ID variable held in `idCol` doesn't matter to `ds.lexis`. Provided every individual appears only once in the original data set (before expansion) the order does not matter because `ds.lexis` works on its unique numeric vector that is allocated from `1:M` (where there are `M` individuals) in whatever order they appear in the original dataset.

in `entryCol` argument rather than using a total survival time variable to identify the intervals to which any given individual is exposed, `ds.lexis` requires an initial entry time and a final exit time. If the data you wish to expand contain only a total survival time variable and every individual starts follow-up at time 0, the entry times should all be specified as zero, and the exit times as the total survival time. So, `entryCol` should either be the name of the column holding the entry time of each individual or else if no `entryCol` is specified it will be defaulted to zero anyway and put into a variable called `starttime` in the expanded data set.

In `exitCol` argument, if the entry times (`entryCol`) are set, or defaulted, to zero, the `exitCol` variable should contain the total survival times.

If `variables` argument is not set (is null) but the `data` argument is set, the expanded data set will contain all variables in the data frame identified by the `data` argument. If neither the `data` or `variables` arguments are set, the expanded data set will only include the ID, exposure time and failure/censoring status variables which may still be useful for plotting survival data once these become available.

This function is particularly meant to be used in preparing data for a piecewise regression analysis (PAR). Although the time intervals have to be pre-specified and are arbitrary, even a vaguely reasonable set of time intervals will give results very similar to a Cox regression analysis. The key issue is to choose survival intervals such that the baseline hazard (risk of death/disease/failure) within each interval is reasonably constant while the baseline hazard can vary freely between intervals. Even

if the choice of intervals is very poor the ultimate results are typically qualitatively similar to Cox regression. Increasing the number of intervals will inevitably improve the approximation to the true baseline hazard, but the addition of many more unnecessary time intervals slows the analysis and can become disclosive and yet will not improve the fit of the model.

If the number of failures in one or more periods in a given study is less than the specified disclosure filter determining minimum acceptable cell size in a table (`nfilter.tab`) then the expanded data frame is not created in that study, and a study-side message to this effect is made available in that study via `ds.message()` function.

Server functions called: `lexisDS1`, `lexisDS2` and `lexisDS3`

Value

`ds.lexis` returns to the server-side a data frame for each study with the expanded version of the input table.

Author(s)

DataSHIELD Development Team

See Also

`ds.glm` for generalized linear models.

Examples

```
## Not run:

 ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  # Example 1: Fitting GLM for survival analysis
  # For this analysis we need to load survival data from the server

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
  logindata <- builder$build()
```

```
# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

#Example 1: Create the expanded data frame.
#The survival time intervals are to be 0<t<=2.5; 2.5<t<=5.0, 5.0<t<=7.5,
#up to the final interval of duration 2.5
#that includes the maximum survival time.

ds.lexis(data = "D",
         intervalWidth = 2.5,
         idCol = "D$id",
         entryCol = "D$starttime",
         exitCol = "D$endtime",
         statusCol = "D$cens",
         expandDF = "EM.new",
         datasources = connections)

#Confirm that the expanded data frame has been ceated
ds.ls(datasources = connections)
#Example 2: Create the expanded data frame.
#The survival time intervals are to be 0<t<=1; 1<t<=2.0, 2.0<t<=5.0, 5.0<t<=11.0,

ds.lexis(data = "D",
         intervalWidth = c(1,1,3,6),
         idCol = "D$id",
         entryCol = "D$starttime",
         exitCol = "D$endtime",
         statusCol = "D$cens",
         expandDF = "EM.new2",
         datasources = connections)

#Confirm expanded dataframe created
ds.ls(datasources = connections)

## End(Not run)
```

`ds.list` *Constructs a list of objects in the server-side*

Description

This is similar to the R function `list`.

Usage

```
ds.list(x = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x	a character string specifying the names of the objects to coerce into a list.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default list.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

If the objects to coerce into a list are for example vectors held in a matrix or a data frame the names of the elements in the list are the names of columns.

Server function called: listDS

Value

ds.list returns a list of objects for each study that is stored on the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

 # combine the 'LAB_TSC' and 'LAB_HDL' variables into a list
```

```
 myobjects <- c('D$LAB_TSC', 'D$LAB_HDL')
 ds.list(x = myobjects,
         newobj = "new.list",
         datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.listClientsideFunctions` *Lists client-side functions*

Description

Lists all current client-side functions

Usage

```
ds.listClientsideFunctions()
```

Details

This function operates by directly interrogating the R objects stored in the input client packages and objects of name starting with `ds.` character in `.GlobalEnv`.

This function does not call any server-side function.

Value

`ds.listClientsideFunctions` returns a list containing all server-side functions.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  #Library with all DataSHIELD functions
  require('dsBaseClient')

  #Visualise all functions
  ds.listClientsideFunctions()


## End(Not run)
```

ds.listDisclosureSettings
Lists disclosure settings

Description

Lists current values for disclosure control filters in all data repository servers.

Usage

```
ds.listDisclosureSettings(datasources = NULL)
```

Arguments

datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function lists out the current values of the eight disclosure filters in each of the data repository servers specified by datasources argument.

The eight filters are explained below:

(1) nfilter.tab, the minimum non-zero cell count allowed in any cell if a contingency table is to be returned. This applies to one dimensional and two dimensional tables of counts tabulated across one or two factors and to tables of a mean of a quantitative variable tabulated across a factor. Default usually set to 3 but a value of 1 (no limit) may be necessary, particularly if low cell counts are highly probable such as when working with rare diseases. Five is also a justifiable choice to replicate the most common threshold rule imposed by data releasers worldwide, but it should be recognised that many census providers are moving to ten - but the formal justification of this is little more than 'it is safer' and everybody is scared of something going wrong - in practice it is very easy to get around any block and so it is debatable whether the scientific cost outweighs the imposition of any threshold.

(2) nfilter.subset, the minimum non-zero count of observational units (typically individuals) in a subset. Typically defaulted to 3.

(3) nfilter.glm, the maximum number of parameters in a regression model as a proportion of the sample size in a study. If a study has 1000 observational units (typically individuals) being used in a particular analysis then if nfilter.glm is set to 0.33 (its default value) the maximum allowable number of parameters in a model fitted to those data will be 330. This disclosure filter protects against fitting overly saturated models that can be disclosive. The choice of 0.33 is entirely arbitrary.

(4) nfilter.string, the maximum length of a string argument if that argument is to be subject to testing of its length. Default value 80. The aim of this nfilter is to make it difficult for hackers to find a way to embed malicious code in a valid string argument that is actively interpreted.

(5) nfilter.string, Short to be used when a string must be specified but that when valid that string should be short.

(6) nfilter.kNN applies to graphical plots based on working with the k nearest neighbours of each point. nfilter.kNN specifies the minimum allowable value for the number of nearest neighbours used, typically defaulted to 3.

(7) nfilter.levels specifies the maximum number of unique levels of a factor variable that can be disclosed to the client. In the absence of this filter a user can convert a numeric variable to a factor and see its unique levels which are all the distinct values of the numeric vector. To prevent such disclosure we set this threshold to 0.33 which ensures that if a factor has unique levels more than the 33

(8) nfilter.noise specifies the minimum level of noise added in some variables mainly used for data visualizations. The default value is 0.25 which means that the noise added to a given variable, follows a normal distribution with zero mean and variance equal to 25 variance of the given variable. Any value greater than this threshold can reduce the risk of disclosure.

Server function called: listDisclosureSettingsDS

Value

ds.listDisclosureSettings returns a list containing the current settings of the nfilters in each study specified.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()
```

```
# Log onto the remote Opal training servers
connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

# Call to list current disclosure settings in all data repository servers

ds.listDisclosureSettings(datasources = connections)

# Restrict call to list disclosure settings only to the first, or second DS connection (study)

ds.listDisclosureSettings(datasources = connections[1])
ds.listDisclosureSettings(datasources = connections[2])

 # Clear the Datashield R sessions and logout
 datashield.logout(connections)

## End(Not run)
```

`ds.listServersideFunctions` *Lists server-side functions*

Description

Lists all current server-side functions

Usage

```
ds.listServersideFunctions(datasources = NULL)
```

Arguments

`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Uses `datashield.methods` function from `DSI` package to list all assign and aggregate functions on the available data repository servers. The only choice of arguments is in `datasources`; i.e. which studies to interrogate. Once the studies have been selected `ds.listServersideFunctions` lists all assign functions for all of these studies and then all aggregate functions for all of them.

This function does not call any server-side function.

Value

`ds.listServersideFunctions` returns to the client-side a list containing all server-side functions separately for each study. Firstly lists assign and then aggregate functions.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # List server-side functions

  ds.listServersideFunctions(datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.lmerSLMA *Fits Linear Mixed-Effect model via Study-Level Meta-Analysis*

Description

ds.lmerSLMA fits a Linear Mixed-Effects Model (lme) - can include both fixed and random-effects - on data from one or multiple sources with pooling via SLMA (Study-Level Meta-Analysis)

Usage

```
ds.lmerSLMA(
  formula = NULL,
  offset = NULL,
  weights = NULL,
  combine.with.metafor = TRUE,
  dataName = NULL,
  checks = FALSE,
  datasources = NULL,
  REML = TRUE,
  control_type = NULL,
  control_value = NULL,
  optimizer = NULL,
  verbose = 0,
  notify.of.progress = FALSE,
  assign = FALSE,
  newobj = NULL
)
```

Arguments

formula	an object of class formula describing the model to be fitted. For more information see **Details**.
offset	a character string specifying the name of a variable to be used as an offset.
weights	a character string specifying the name of a variable containing prior regression weights for the fitting process.
combine.with.metafor	logical. If TRUE the estimates and standard errors for each regression coefficient are pooled across studies using random-effects meta-analysis under maximum likelihood (ML), restricted maximum likelihood (REML) or fixed-effects meta-analysis (FE). Default TRUE.
dataName	a character string specifying the name of an (optional) data frame that contains all of the variables in the LME formula. For more information see **Details**.
checks	logical. If TRUE ds.lmerSLMA checks the structural integrity of the model. Default FALSE. For more information see **Details**.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa
REML	logical. If TRUE the REstricted Maximum Likelihood (REML) is used for parameter optimization. If FALSE the parameters are optimized using standard ML (maximum likelihood). Default TRUE. For more information see **Details**.
control_type	an optional character string vector specifying the nature of a parameter (or parameters) to be modified in the convergence control options which can be viewed or modified via the lmerControl function of the package lme4. For more information see **Details**.
control_value	numeric representing the new value which you want to allocate the control parameter corresponding to the control-type. For more information see **Details**.

optimizer — specifies the parameter optimizer that lmer should use. For more information see **Details**.

verbose — an integer value. If $verbose > 0$ the output is generated during the optimization of the parameter estimates. If $verbose > 1$ the output is generated during the individual penalized iteratively reweighted least squares (PIRLS) steps. Default verbose value is 0 which means no additional output.

notify.of.progress — specifies if console output should be produced to indicate progress. Default FALSE.

assign — a logical, indicates whether the function will call a second server-side function (an assign) in order to save the regression outcomes (i.e. a lmerMod object) on each server. Default FALSE.

newobj — a character string specifying the name of the object to which the lmerMod object representing the model fit on the serverside in each study is to be written. This argument is used only when the argument assign is set to TRUE. If no <newobj> argument is specified, the output object defaults to "new.lmer.obj".

Details

ds.lmerSLMA fits a Linear Mixed Effects Model (lme) - can include both fixed and random effects - on data from single or multiple sources.

This function is similar to lmer function from lme4 package in native R.

When there are multiple data sources, the LME is fitted to convergence in each data source independently. The estimates and standard errors returned to the client-side which enable cross-study pooling using Study-Level Meta-Analysis (SLMA). The SLMA used by default metafor package but as the SLMA occurs on the client-side (a standard R environment), the user can choose any approach to meta-analysis. Additional information about fitting LMEs using the lmer function can be obtained using R help for lmer and the lme4 package.

In formula most shortcut notation allowed by lmer() function is also allowed by ds.lmerSLMA. Many LMEs can be fitted very simply using a formula like: $y\ a + b + (1|c)$ which simply means fit an LME with y as the outcome variable with a and b as fixed effects, and c as a random effect or grouping factor.

It is also possible to fit models with random slopes by specifying a model such as $y\ a + b + (1 + b|c)$ where the effect of b can vary randomly between groups defined by c. Implicit nesting can be specified with formulae such as $y\ a + b + (1|c/d)$ or $y\ a + b + (1|c) + (1|c : d)$.

The dataName argument avoids you having to specify the name of the data frame in front of each covariate in the formula. For example, if the data frame is called DataFrame you avoid having to write: $DataFrame\$y\ DataFrame\$a + DataFrame\$b + (1|DataFrame\$c)$.

The checks argument verifies that the variables in the model are all defined (exist) on the server-site at every study and that they have the correct characteristics required to fit the model. It is suggested to make checks argument TRUE if an unexplained problem in the model fit is encountered because the running process takes several minutes.

REML can help to mitigate bias associated with the fixed-effects. See help on the lmer() function for more details.

In control_type at present only one such parameter can be modified, namely the tolerance of the convergence criterion to the gradient of the log-likelihood at the maximum likelihood achieved. We

have enabled this because our practical experience suggests that in situations where the model looks to have converged with sensible parameter values but formal convergence is not being declared if we allow the model to be more tolerant to a non-zero gradient the same parameter values are obtained but formal convergence is declared. The default value for the `check.conv.grad` is `0.002`.

`control_value` At present (see `control_type`) the only parameter this can be is the convergence tolerance `check.conv.grad`. In general, models will be identified as having converged more readily if the value set for `check.conv.grad` is increased from its default (`0.002`). Please note that the risk of doing this is that the model is also more likely to be declared as having converged at a local maximum that is not the global maximum likelihood. This will not generally be a problem if the likelihood surface is well behaved but if you have a problem with convergence you might usefully compare all the parameter estimates and standard errors obtained using the default tolerance (`0.002`) even though that has not formally converged with those obtained after convergence using the higher tolerance.

The `optimizer` argument is built in but it won't do anything because there is only one standard optimizer available for lmer - this is the `nloptwrap` optimizer. If users wish to apply a different optimizer - potentially one they have developed themselves - the development team can activate this argument so alternatives can be specified.

Server function called: `lmerSLMADS2`

Value

Many of the elements of the output list returned by `ds.lmerSLMA` are equivalent to those returned by the `lmer()` function in native R. However, potentially disclosive elements such as individual-level residuals and linear predictor values are blocked. In this case, only non-disclosive elements are returned from each study separately.

The list of elements returned by `ds.lmerSLMA` is mentioned below:

`ds.lmerSLMA` returns a list of elements mentioned below separately for each study.

`coefficients`: a matrix with 5 columns:

First : the names of all of the regression parameters (coefficients) in the model

second : the estimated values

third : corresponding standard errors of the estimated values

fourth : the ratio of estimate/standard error

fifth : the p-value treating that as a standardised normal deviate

`CorrMatrix`: the correlation matrix of parameter estimates.

`VarCovMatrix`: the variance-covariance matrix of parameter estimates.

`weights`: the vector (if any) holding regression weights.

`offset`: the vector (if any) holding an offset.

`cov.scaled`: equivalent to `VarCovMatrix`.

`Nmissing`: the number of missing observations in the given study.

`Nvalid`: the number of valid (non-missing) observations in the given study.

`Ntotal`: the total number of observations in the given study (`Nvalid` + `Nmissing`).

`data`: equivalent to input parameter `dataName` (above).

call: summary of key elements of the call to fit the model.

There are a small number of more esoteric items of the information returned by ds.lmerSLMA. Additional information about these can be found in the help file for the lmer() function in the lme4 package.

Once the study-specific output has been returned, the function returns several elements relating to the pooling of estimates across studies via study-level meta-analysis. These are as follows:

input.beta.matrix.for.SLMA: a matrix containing the vector of coefficient estimates from each study.

input.se.matrix.for.SLMA: a matrix containing the vector of standard error estimates for coefficients from each study.

SLMA.pooled.estimates: a matrix containing pooled estimates for each regression coefficient across all studies with pooling under SLMA via random-effects meta-analysis under maximum likelihood (ML), restricted maximum likelihood (REML) or via fixed-effects meta-analysis (FE).

convergence.error.message: reports for each study whether the model converged. If it did not some information about the reason for this is reported.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CLUSTER.CLUSTER_SLO1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CLUSTER.CLUSTER_SLO2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CLUSTER.CLUSTER_SLO3", driver = "OpalDriver")
  logindata <- builder$build()

   #Log onto the remote Opal training servers
   connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")
```

```
  # Select all rows without missing values
  ds.completeCases(x1 = "D", newobj = "D.comp", datasources = connections)

  # Fit the lmer

  ds.lmerSLMA(formula = "BMI ~ incid_rate + diabetes + (1 | Male)",
              dataName = "D.comp",
              datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.log` *Computes logarithms in the server-side*

Description

Computes the logarithms for a specified numeric vector. This function is similar to the R `log` function. by default natural logarithms.

Usage

```
ds.log(x = NULL, base = exp(1), newobj = NULL, datasources = NULL)
```

Arguments

`x`	a character string providing the name of a numerical vector.
`base`	a positive number, the base for which logarithms are computed. Default `exp(1)`.
`newobj`	a character string that provides the name for the output variable that is stored on the server-side. Default `log.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Server function called: `log`

Value

`ds.log` returns a vector for each study of the transformed values for the numeric vector specified in the argument `x`. The created vectors are stored in the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")

  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Calculating the log value of the 'PM_BMI_CONTINUOUS' variable

  ds.log(x = "D$PM_BMI_CONTINUOUS",
         base = exp(2),
         newobj = "log.PM_BMI_CONTINUOUS",
         datasources = connections[1]) #only the first Opal server is used (study1)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.look` *Performs direct call to a server-side aggregate function*

Description

The function `ds.look` can be used to make a direct call to a server-side aggregate function more simply than using the `datashield.aggregate` function.

Usage

```
ds.look(toAggregate = NULL, checks = FALSE, datasources = NULL)
```

Arguments

toAggregate	a character string specifying the function call to be made. For more information see **Details**.
checks	logical. If TRUE the optional checks are undertaken. Default FALSE to save time.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The ds.look and datashield.aggregate functions are generally only recommended for experienced developers. For example, the toAggregate argument has to be expressed in the same form that the server-side function would usually expect from its client-side pair. For example: ds.look("table1DDS(female)") works. But, if you express this as ds.look("table1DDS('female')") it won't work because although when you call this same function using its client-side function you write ds.table1D('female') the inverted commas are stripped off during processing by the client-side function so the call to the server-side does not contain inverted commas.

Apart from during development work (e.g. before a client-side function has been written) it is almost always easier and less error-prone to call a server-side function using its client-side pair.

The function is a wrapper for the DSI package function datashield.aggregate.

Value

the output from the specified server-side aggregate function to the client-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
  builder$append(server = "study2",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
  builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Calculate the length of a variable using the server-side function

  ds.look(toAggregate = "lengthDS(D$age.60)",
          checks = FALSE,
          datasources = connections)

  #Calculate the column names of "D" object using the server-side function

  ds.look(toAggregate = "colnames(D)",
          checks = FALSE,
          datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.ls` *lists all objects on a server-side environment*

Description

creates a list of the names of all of the objects in a specified serverside environment.

Usage

```
ds.ls(
  search.filter = NULL,
  env.to.search = 1L,
  search.GlobalEnv = TRUE,
  datasources = NULL
)
```

Arguments

`search.filter`	character string (potentially including * symbol) specifying the filter for the object name that you want to find in the environment. For more information see **Details**.

`env.to.search` an integer (e.g. in `2` or `2L` format) specifying the position in the search path of the environment to be explored. `1L` is the current active analytic environment on the server-side and is the default value of `env.to.search`. For more information see **Details**.

`search.GlobalEnv` Logical. If TRUE, `ds.ls` will list all objects in the `.GlobalEnv` R environment on the server-side. If FALSE and if `env.to.search` is also set as a valid integer, `ds.ls` will list all objects in the server-side R environment identified by `env.to.search` in the search path. For more information see **Details**.

`datasources` a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

When running analyses one may want to know the objects already generated. This request is not disclosive as it only returns the names of the objects and not their contents.

By default, objects in DataSHIELD's Active Serverside Analytic Environment (`.GlobalEnv`) will be listed. This is the environment that contains all of the objects that server-side DataSHIELD is using for the main analysis or has written out to the server-side during the process of managing or undertaking the analysis (variables, scalars, matrices, data frames, etc).

The environment to explore is specified by the argument `env.to.search` (i.e. environment to search) to an integer value. The default environment which R names as `.GlobalEnv` is set by specifying `env.to.search = 1` or `1L` (`1L` is just an explicit way of writing the integer `1`).

If the `search.GlobalEnv` argument is set to TRUE the `env.to.search` parameter is set to `1L` regardless of what value it is set in the call or if it is set to NULL. So, if `search.GlobalEnv` is set to TRUE, `ds.ls` will automatically search the `.GlobalEnv` R environment on the server-side which contains all of the variables, data frames and other objects read in at the start of the analysis, as well as any new objects of any sort created using DataSHIELD assign functions.

Other server-side environments contain other objects. For example, environment `2L` contains the functions loaded via the native R stats package and `6L` contains the standard list of datasets built into R. By default `ds.ls` will return a list of ALL of the objects in the environment specified by the `env.to.search` argument but you can specify search filters including `*` wildcards using the `search.filter` argument.

In `search.filter` you can use the symbol `*` to find all the object that contains the specified characters. For example, `search.filter = "Sd2*"` will list the names of all objects in the specified environment with names beginning capital S, lower case d and number 2. Similarly, `search.filter="*.ID"` will return all objects with names ending with `.ID`, for example `Study.ID`. If a value is not specified for the `search.filter` argument or it is set as NULL, the names of all objects in the specified environment will be returned.

Server function called: `lsDS`.

Value

`ds.ls` returns to the client-side a list containing:
(1) the name/details of the server-side R environment which `ds.ls` has searched;
(2) a vector of character strings giving the names of all objects meeting the naming criteria specified

by the argument search.filter in this specified R server-side environment;
(3) the nature of the search filter string as it was applied.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Example 1: Obtain the list of  all objects on a server-side environment

  ds.ls(datasources = connections)

  #Example 2: Obtain the list of all objects that contain "var" character in the name
  #Create in the server-side variables with "var" character in the name

  ds.assign(toAssign = "D$LAB_TSC",
            newobj = "var.LAB_TSC",
            datasources = connections)
  ds.assign(toAssign = "D$LAB_TRIG",
            newobj = "var.LAB_TRIG",
            datasources = connections)
  ds.assign(toAssign = "D$LAB_HDL",
            newobj = "var.LAB_HDL",
            datasources = connections)
```

```
  ds.ls(search.filter = "var*",
        env.to.search = 1L,
        search.GlobalEnv = TRUE,
        datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.lspline` *Basis for a piecewise linear spline with meaningful coefficients*

Description

This function is based on the native R function `lspline` from the `lspline` package. This function computes the basis of piecewise-linear spline such that, depending on the argument marginal, the coefficients can be interpreted as (1) slopes of consecutive spline segments, or (2) slope change at consecutive knots.

Usage

```
ds.lspline(
  x,
  knots = NULL,
  marginal = FALSE,
  names = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`x`	the name of the input numeric variable
`knots`	numeric vector of knot positions
`marginal`	logical, how to parametrise the spline, see Details
`names`	character, vector of names for constructed variables
`newobj`	a character string that provides the name for the output variable that is stored on the data servers. Default `lspline.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

If marginal is FALSE (default) the coefficients of the spline correspond to slopes of the consecutive segments. If it is TRUE the first coefficient correspond to the slope of the first segment. The consecutive coefficients correspond to the change in slope as compared to the previous segment.

Value

an object of class "lspline" and "matrix", which its name is specified by the newobj argument (or its default name "lspline.newobj"), is assigned on the serverside.

Author(s)

Demetris Avraam for DataSHIELD Development Team

---

ds.make *Calculates a new object in the server-side*

Description

This function defines a new object in the server-side via an allowed function or an arithmetic expression.

ds.make function is equivalent to ds.assign, but runs slightly faster.

Usage

```
ds.make(toAssign = NULL, newobj = NULL, datasources = NULL)
```

Arguments

toAssign	a character string specifying the function or the arithmetic expression.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default make.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

If the new object is created successfully, the function will verify its existence on the required servers. Please note there are certain modes of failure where it is reported that the object has been created but it is not there. This reflects a failure in the processing of some sort and warrants further exploration of the details of the call to ds.make and the variables/objects which it invokes.

TROUBLESHOOTING: please note we have recently identified an error that makes ds.make fail and DataSHIELD crash.

The error arises from a call such as ds.make(toAssign = '5.3 + beta*xvar', newobj = 'predvals'). This is a typical call you may make to get the predicted values from a simple linear regression model where a y variable is regressed against an x variable (xvar) where the estimated regression intercept is 5.3 and beta is the estimated regression slope.

This call appears to fail because in interpreting the arithmetic function which is its first argument it first encounters the (length 1) scalar 5.3 and when it then encounters the xvar vector which has more than one element it fails - apparently because it does not recognise that you need to replicate

the 5.3 value the appropriate number of times to create a vector of length equal to xvar with each value equal to 5.3.

There are two work-around solutions here:

(1) explicitly create a vector of appropriate length with each value equal to 5.3. To do this there is a useful trick. First identify a convenient numeric variable with no missing values (typically a numeric individual ID) let us call it indID equal in length to xvar (xvar may include NAs but that doesn't matter provided indID is the same total length). Then issue the call ds.make(toAssign = 'indID-indID+1',newobj = 'ONES'). This creates a vector of ones (called ONES) in each source equal in length to the indID vector in that source. Then issue the second call ds.make(toAssign = 'ONES*5.3',newobj = 'vect5.3') which creates the required vector of length equal to xvar with all elements 5.3. Finally, you can now issue a modified call to reflect what was originally needed: ds.make(toAssign = 'vect5.3+beta*xvar', 'predvals').

(2) Alternatively, if you simply swap the original call around: ds.make(toAssign = '(beta*xvar)+5.3', newobj = 'predvals') the error seems also to be circumvented. This is presumably because the first element of the arithmetic function is of length equal to xvar and it then knows to replicate the 5.3 that many times in the second part of the expression.

The second work-around is easier, but it is worth knowing about the first trick because creating a vector of ones of equal length to another vector can be useful in other settings. Equally the call: ds.make(toAssign = 'indID-indID',newobj = 'ZEROS') to create a vector of zeros of that same length may also be useful.

Server function : messageDS

The ds.make function is a wrapper for the DSI package function datashield.assign

Value

ds.make returns the new object which is written to the server-side. Also a validity message is returned to the client-side indicating whether the new object has been correctly created at each source.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
```

```
                  user = "administrator", password = "datashield_test&",
                  table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
  builder$append(server = "study2",
                  url = "http://192.168.56.100:8080/",
                  user = "administrator", password = "datashield_test&",
                  table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
  builder$append(server = "study3",
                  url = "http://192.168.56.100:8080/",
                  user = "administrator", password = "datashield_test&",
                  table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

##Example 1: arithmetic operators

ds.make(toAssign = "D$age.60 + D$bmi.26",
        newobj = "exprs1",
        datasources = connections)

ds.make(toAssign = "D$noise.56 + D$pm10.16",
        newobj = "exprs2",
        datasources = connections)

ds.make(toAssign = "(exprs1*exprs2)/3.2",
        newobj = "result.example1",
        datasources = connections)

##Example 2: miscellaneous operators within functions

ds.make(toAssign = "(D$female)^2",
        newobj = "female2",
       datasources = connections)

ds.make(toAssign = "(2*D$female)+(D$log.surv)-(female2*2)",
        newobj = "output.test.1",
        datasources = connections)

ds.make(toAssign = "exp(output.test.1)",
        newobj = "output.test",
       datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.matrix *Creates a matrix on the server-side*

Description

Creates a matrix on the server-side with dimensions specified by `nrows.scalar` and `ncols.scalar` arguments and assigns the values of all its elements based on the `mdata` argument.

Usage

```
ds.matrix(
  mdata = NA,
  from = "clientside.scalar",
  nrows.scalar = NULL,
  ncols.scalar = NULL,
  byrow = FALSE,
  dimnames = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`mdata`	a character string specifying the name of a server-side scalar or vector. Also, a numeric value representing a scalar specified from the client-side can be specified Zeros, negative values and NAs are all allowed. For more information see **Details**.
`from`	a character string specifying the source and nature of `mdata`. This can be set as `"serverside.vector"`, `"serverside.scalar"` or `"clientside.scalar"`. Default `"clientside.scalar"`.
`nrows.scalar`	an integer or a character string that specifies the number of rows in the matrix to be created. For more information see **Details**.
`ncols.scalar`	an integer or a character string that specifies the number of columns in the matrix to be created.
`byrow`	logical. If TRUE and `mdata` is a vector the matrix created should be filled row by row. If FALSE the matrix created should be filled column by column. Default = FALSE.
`dimnames`	a list of length 2 giving the row and column names respectively.
`newobj`	a character string that provides the name for the output variable that is stored on the data servers. Default `matrix.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function is similar to the R native function `matrix()`.

If in the `mdata` argument a vector is specified this should have the same length as the total number of elements in the matrix. If this is not TRUE the values in `mdata` will be used repeatedly until all elements in the matrix are full. If `mdata` argument is a scalar, all elements in the matrix will take that value.

In the nrows.scalar argument can be a character string specifying the name of a server-side scalar. For example, if a server-side scalar named ss.scalar exists and holds the value 23, then by specifying nrows.scalar = "ss.scalar", the matrix created will have 23 rows. Also this argument can be a numeric value from the client-side. The same rules are applied to ncols.scalar argument but in this case the column numbers are specified. In both arguments a zero, negative, NULL or missing value is not permitted.

Server function called: matrixDS

Value

ds.matrix returns the created matrix which is written on the server-side. In addition, two validity messages are returned indicating whether the new matrix has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Example 1: create a matrix with -13 value in all elements

  ds.matrix(mdata = -13,
            from = "clientside.scalar",
```

```
          nrows.scalar = 3,
          ncols.scalar = 8,
          newobj = "cs.block",
          datasources = connections)

#Example 2: create a matrix of missing values

ds.matrix(NA,
          from = "clientside.scalar",
          nrows.scalar = 4,
          ncols.scalar = 5,
          newobj = "cs.block.NA",
          datasources = connections)

#Example 3: create a matrix using a server-side vector
#create a vector in the server-side

ds.rUnif(samp.size = 45,
         min = -10.5,
         max = 10.5,
         newobj = "ss.vector",
         seed.as.integer = 8321,
         force.output.to.k.decimal.places = 0,
         datasources = connections)

ds.matrix(mdata = "ss.vector",
          from = "serverside.vector",
          nrows.scalar = 5,
          ncols.scalar = 9,
          newobj = "sv.block",
          datasources = connections)

#Example 4: create a matrix using a server-side vector and specifying
#the row a column names

ds.rUnif(samp.size = 9,
         min = -10.5,
         max = 10.5,
         newobj = "ss.vector.9",
         seed.as.integer = 5575,
         force.output.to.k.decimal.places = 0,
         datasources = connections)

ds.matrix(mdata = "ss.vector.9",
          from = "serverside.vector",
          nrows.scalar = 5,
          ncols.scalar = 9,
          byrow = TRUE,
          dimnames = list(c("a","b","c","d","e")),
          newobj = "sv.block.9.dimnames1",
          datasources = connections)
```

```
  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.matrixDet *Calculates de determinant of a matrix in the server-side*

Description

Calculates the determinant of a square matrix that is written on the server-side. This operation is only possible if the number of columns and rows of the matrix are the same.

Usage

```
ds.matrixDet(M1 = NULL, newobj = NULL, logarithm = FALSE, datasources = NULL)
```

Arguments

M1	a character string specifying the name of the matrix.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default matrixdet.newobj.
logarithm	logical. If TRUE the logarithm of the modulus of the determinant is calculated. Default FALSE.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Calculates the determinant of a square matrix on the server-side. This function is similar to the native R determinant function.

Server function called: matrixDetDS2

Value

ds.matrixDet returns the determinant of an existing matrix on the server-side. The created new object is stored on the server-side. Also, two validity messages are returned indicating whether the matrix has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Create the matrix in the server-side

  ds.rUnif(samp.size = 9,
           min = -10.5,
           max = 10.5,
           newobj = "ss.vector.9",
           seed.as.integer = 5575,
           force.output.to.k.decimal.places = 0,
           datasources = connections)

  ds.matrix(mdata = "ss.vector.9",
            from = "serverside.vector",
            nrows.scalar = 9,ncols.scalar = 9,
            byrow = TRUE,
            newobj = "matrix",
            datasources = connections)

  #Calculate the determinant of the matrix

  ds.matrixDet(M1 = "matrix",
               newobj = "matrixDet",
               logarithm = FALSE,
               datasources = connections)
```

```
  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.matrixDet.report` *Returns matrix determinant to the client-side*

Description

Calculates the determinant of a square matrix and returns the result to the client-side

Usage

```
ds.matrixDet.report(M1 = NULL, logarithm = FALSE, datasources = NULL)
```

Arguments

`M1`	a character string specifying the name of the matrix.
`logarithm`	logical. If TRUE the logarithm of the modulus of the determinant is calculated. Default FALSE.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Calculates and returns to the client-side the determinant of a square matrix on the server-side. This function is similar to the native R `determinant` function. This operation is only possible if the number of columns and rows of the matrix are the same.

Server function called: `matrixDetDS1`

Value

`ds.matrixDet.report` returns to the client-side the determinant of a matrix that is stored on the server-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Create the matrix in the server-side

  ds.rUnif(samp.size = 9,
           min = -10.5,
           max = 10.5,
           newobj = "ss.vector.9",
           seed.as.integer = 5575,
           force.output.to.k.decimal.places = 0,
           datasources = connections)

  ds.matrix(mdata = "ss.vector.9",
            from = "serverside.vector",
            nrows.scalar = 9,ncols.scalar = 9,
            byrow = TRUE,
            newobj = "matrix",
            datasources = connections)

  #Calculate the determinant of the matrix

  ds.matrixDet.report(M1 = "matrix",
                      logarithm = FALSE,
                      datasources = connections)

  # clear the Datashield R sessions and logout
```

```
  datashield.logout(connections)

## End(Not run)
```

---

ds.matrixDiag *Calculates matrix diagonals in the server-side*

Description

Extracts the diagonal vector from a square matrix or creates a diagonal matrix based on a vector or a scalar value on the server-side.

Usage

```
ds.matrixDiag(
  x1 = NULL,
  aim = NULL,
  nrows.scalar = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

x1	a character string specifying the name of a server-side scalar or vector. Also, a numeric value or vector specified from the client-side can be specified. This argument depends on the value specified in aim. For more information see **Details**.
aim	a character string specifying the behaviour of the function. This can be set as: "serverside.vector.2.matrix", "serverside.scalar.2.matrix", "serverside.matrix.2.vecto "clientside.vector.2.matrix" or "clientside.scalar.2.matrix". For more information see **Details**.
nrows.scalar	an integer specifying the dimensions of the matrix note that the matrix is square (same number of rows and columns). If this argument is not specified the matrix dimensions are defined by the length of the vector. For more information see **Details**.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default matrixdiag.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The function behaviour is different depending on the value specified in the aim argument:
(1) If aim = "serverside.vector.2.matrix" the function takes a server-side vector and writes out a square matrix with the vector as its diagonal and all off-diagonal values = 0. The dimensions

of the output matrix are determined by the length of the vector. If the vector length is k, the output matrix has k rows and k columns.
(2) If aim = "serverside.scalar.2.matrix" the function takes a server-side scalar and writes out a square matrix with all diagonal values equal to the value of the scalar and all off-diagonal values = 0. The dimensions of the square matrix are determined by the value of the nrows.scalar argument.
(3) If aim = "serverside.matrix.2.vector" the function takes a square server-side matrix and extracts its diagonal values as a vector which is written to the server-side.
(4) If aim = "clientside.vector.2.matrix" the function takes a vector specified on the client-side and writes out a square matrix to the server-side with the vector as its diagonal and all off-diagonal values = 0. The dimensions of the output matrix are determined by the length of the vector.
(5) If aim = "clientside.scalar.2.matrix" the function takes a scalar specified on the client-side and writes out a square matrix with all diagonal values equal to the value of the scalar. The dimensions of the square matrix are determined by the value of the nrows.scalar argument.

If x1 is a vector and the nrows.scalar is set as k, the vector will be used repeatedly to fill up the diagonal. For example, the vector is of length 7 and nrows.scalar = 18, a square diagonal matrix with 18 rows and 18 columns will be created.

Server function called: matrixDiagDS

Value

ds.matrixDiag returns to the server-side the square matrix diagonal. Also, two validity messages are returned indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

#Example 1: Create a square matrix with the server-side vector as its diagonal
#and all the other values = 0

# Create a vector in the server-side

ds.rUnif(samp.size = 9,
         min = -10.5,
         max = 10.5,
         newobj = "ss.vector.9",
         seed.as.integer = 5575,
         force.output.to.k.decimal.places = 0,
         datasources = connections)


#Calculate the diagonal of the matrix

ds.matrixDiag(x1 = "ss.vector.9",
              aim = "serverside.vector.2.matrix",
              nrows.scalar = NULL,
              newobj = "matrix.diag1",
              datasources = connections)

#Example 2: Create a square matrix with the server-side scalar as all diagonal values
#and all the other values = 0

#Create a scalar in the server-side

ds.rUnif(samp.size = 1,
         min = -10.5,
         max = 10.5,
         newobj = "ss.scalar",
         seed.as.integer = 5575,
         force.output.to.k.decimal.places = 0,
         datasources = connections)

#Calculate the diagonal of the matrix

ds.matrixDiag(x1 = "ss.scalar",
              aim = "serverside.scalar.2.matrix",
              nrows.scalar = 4,
              newobj = "matrix.diag2",
              datasources = connections)

#Example 3: Create a vector that contains the server-side matrix diagonal values
```

```
#Create a matrix in the server-side

ds.matrix(mdata = 10,
          from = "clientside.scalar",
          nrows.scalar = 3,
          ncols.scalar = 8,
          newobj = "ss.matrix",
          datasources = connections)

#Extract the diagonal of the matrix

ds.matrixDiag(x1 = "ss.matrix",
              aim = "serverside.matrix.2.vector",
              nrows.scalar = NULL,
              newobj = "vector.diag3",
              datasources = connections)

#Example 4: Create a square matrix with the client-side vector as a diagonal
#and all the other values = 0

ds.matrixDiag(x1 = c(2,6,9,10),
              aim = "clientside.vector.2.matrix",
              nrows.scalar = NULL,
              newobj = "matrix.diag4",
              datasources = connections)

#Example 5: Create a square matrix with the client-side scalar as all diagonal values
#and all the other values = 0

ds.matrixDiag(x1 = 4,
              aim = "clientside.scalar.2.matrix",
              nrows.scalar = 5,
              newobj = "matrix.diag5",
              datasources = connections)


 # clear the Datashield R sessions and logout
 datashield.logout(connections)

## End(Not run)
```

`ds.matrixDimnames` *Specifies the dimnames of the server-side matrix*

Description

Adds the row names, the column names or both to a matrix on the server-side.

Usage

```
ds.matrixDimnames(
  M1 = NULL,
  dimnames = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

M1	a character string specifying the name of a server-side matrix.
dimnames	a list of length 2 giving the row and column names respectively. An empty list is treated as NULL.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default matrixdimnames.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the native R dimnames function.

Server function called: matrixDimnamesDS

Value

ds.matrixDimnames returns to the server-side the matrix with specified row and column names. Also, two validity messages are returned to the client-side indicating the new object that has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


#Example 1: Set the row and column names of a server-side matrix

#Create the server-side vector

ds.rUnif(samp.size = 9,
         min = -10.5,
         max = 10.5,
         newobj = "ss.vector.9",
         seed.as.integer = 5575,
         force.output.to.k.decimal.places = 0,
         datasources = connections)

#Create the server-side matrix

ds.matrix(mdata = "ss.vector.9",
          from = "serverside.vector",
          nrows.scalar = 3,
          ncols.scalar = 4,
          byrow = TRUE,
          newobj = "matrix",
          datasources = connections)

#Specify the column and row names of the matrix

ds.matrixDimnames(M1 = "matrix",
                  dimnames = list(c("a","b","c"),c("a","b","c","d")),
                  newobj = "matrix.dimnames",
                  datasources = connections)

# clear the Datashield R sessions and logout
datashield.logout(connections)

## End(Not run)
```

---

ds.matrixInvert *Inverts a server-side square matrix*

Description

Inverts a square matrix and writes the output to the server-side

Usage

```
ds.matrixInvert(M1 = NULL, newobj = NULL, datasources = NULL)
```

Arguments

M1	A character string specifying the name of the matrix to be inverted.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default matrixinvert.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This operation is only possible if the number of columns and rows of the matrix are the same and it is non-singular-positive definite (e.g. there is no row or column that is all zeros).

Server function called: matrixInvertDS

Value

ds.matrixInvert returns to the server-side the inverts square matrix. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
```

```
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


#Example 1: Invert the server-side matrix

#Create the server-side vector

ds.rUnif(samp.size = 9,
         min = -10.5,
         max = 10.5,
         newobj = "ss.vector.9",
         seed.as.integer = 5575,
         force.output.to.k.decimal.places = 0,
         datasources = connections)

#Create the server-side matrix

ds.matrix(mdata = "ss.vector.9",
          from = "serverside.vector",
          nrows.scalar = 3,
          ncols.scalar = 4,
          byrow = TRUE,
          newobj = "matrix",
          datasources = connections)

#Invert the matrix

ds.matrixInvert(M1 = "matrix",
                newobj = "matrix.invert",
                datasources = connections)

# clear the Datashield R sessions and logout
datashield.logout(connections)

## End(Not run)
```

`ds.matrixMult` *Calculates tow matrix multiplication in the server-side*

Description

Calculates the matrix product of two matrices and writes output to the server-side.

Usage

```
ds.matrixMult(M1 = NULL, M2 = NULL, newobj = NULL, datasources = NULL)
```

Arguments

M1	a character string specifying the name of the first matrix.
M2	a character string specifying the name of the second matrix.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default matrixmult.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Undertakes standard matrix multiplication wherewith input matrices A and B with dimensions A: m x n and B: n x p the output matrix C has dimensions m x p. This calculation is only valid if the number of columns of A is the same as the number of rows of B.

Server function called: matrixMultDS

Value

ds.matrixMult returns to the server-side the result of the two matrix multiplication. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


  #Example 1: Multiplicate two server-side matrix

  #Create the server-side vector

  ds.rUnif(samp.size = 9,
           min = -10.5,
           max = 10.5,
           newobj = "ss.vector.9",
           seed.as.integer = 5575,
           force.output.to.k.decimal.places = 0,
           datasources = connections)

  #Create the server-side matrixes

  ds.matrix(mdata = "ss.vector.9",#using the created vector
            from = "serverside.vector",
            nrows.scalar = 5,
            ncols.scalar = 4,
            byrow = TRUE,
            newobj = "matrix1",
            datasources = connections)

   ds.matrix(mdata = 10,
             from = "clientside.scalar",
             nrows.scalar = 4,
             ncols.scalar = 6,
             byrow = TRUE,
             newobj = "matrix2",
             datasources = connections)

  #Multiplicate the matrixes

  ds.matrixMult(M1 = "matrix1",
                M2 = "matrix2",
                newobj = "matrix.mult",
                datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.matrixTranspose` *Transposes a server-side matrix*

Description

Transposes a matrix and writes the output to the server-side

Usage

```
ds.matrixTranspose(M1 = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`M1`	a character string specifying the name of the matrix.
`newobj`	a character string that provides the name for the output variable that is stored on the data servers. Default `matrixtranspose.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This operation converts matrix `A` to matrix `C` where element `C[i,j]` of matrix `C` equals element `A[j,i]` of matrix `A`. Matrix `A`, therefore, has the same number of rows as matrix `C` has columns and vice versa.

Server function called: `matrixTransposeDS`

Value

`ds.matrixTranspose` returns to the server-side the transpose matrix. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
builder <- DSI::newDSLoginBuilder()
builder$append(server = "study1",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM1", driver = "OpalDriver")
builder$append(server = "study2",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
               url = "http://192.168.56.100:8080/",
               user = "administrator", password = "datashield_test&",
               table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


#Example 1: Transpose the server-side matrix

#Create the server-side vector

ds.rUnif(samp.size = 9,
         min = -10.5,
         max = 10.5,
         newobj = "ss.vector.9",
         seed.as.integer = 5575,
         force.output.to.k.decimal.places = 0,
         datasources = connections)

#Create the server-side matrix

ds.matrix(mdata = "ss.vector.9",
          from = "serverside.vector",
          nrows.scalar = 3,
          ncols.scalar = 4,
          byrow = TRUE,
          newobj = "matrix",
          datasources = connections)

#Transpose the matrix

ds.matrixTranspose(M1 = "matrix",
                   newobj = "matrix.transpose",
                   datasources = connections)

# clear the Datashield R sessions and logout
datashield.logout(connections)

## End(Not run)
```

`ds.mdPattern` *Display missing data patterns with disclosure control*

Description

This function is a client-side wrapper for the server-side mdPatternDS function. It generates a missing data pattern matrix similar to mice::md.pattern but with disclosure control applied to prevent revealing small cell counts.

Usage

```
ds.mdPattern(x = NULL, type = "split", datasources = NULL)
```

Arguments

`x`	a character string specifying the name of a data frame or matrix on the server-side containing the data to analyze.
`type`	a character string specifying the output type. If 'split' (default), returns separate patterns for each study. If 'combine', attempts to pool patterns across studies.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified, the default set of connections will be used: see `datashield.connections_defa`

Details

The function calls the server-side mdPatternDS function which uses mice::md.pattern to analyze missing data patterns. Patterns with counts below the disclosure threshold (default: nfilter.tab = 3) are suppressed to maintain privacy.

Output Format: - Each row represents a missing data pattern - Pattern counts are shown in row names (e.g., "150", "25") - Columns show 1 if the variable is observed, 0 if missing - Last column shows the total number of missing values per pattern - Last row shows the total number of missing values per variable

Disclosure Control:

Suppressed patterns (count below threshold) are indicated by: - Row name: "suppressed(<N>)" where N is the threshold - All pattern values set to NA - Summary row also suppressed to prevent back-calculation

Pooling Behavior (type='combine'):

When pooling across studies, the function uses a *conservative approach* for disclosure control:

1. Identifies identical missing patterns across studies 2. **EXCLUDES suppressed patterns from pooling** - patterns suppressed in ANY study are not included in the pooled count 3. Sums counts only for non-suppressed identical patterns 4. Re-validates pooled counts against disclosure threshold

Important: This conservative approach means: - Pooled counts may be *underestimates* if some studies had suppressed patterns - This prevents disclosure through subtraction (e.g., if study A shows count=5 and pool shows count=7, one could deduce study B has count=2, violating disclosure) - Different patterns across studies are preserved separately in the pooled result

Value

For type='split': A list with one element per study, each containing:

pattern The missing data pattern matrix for that study

valid Logical indicating if all patterns meet disclosure requirements

message A message describing the validity status

For type='combine': A list containing:

pattern The pooled missing data pattern matrix across all studies

valid Logical indicating if all pooled patterns meet disclosure requirements

message A message describing the validity status

Author(s)

Xavier Escribà montagut for DataSHIELD Development Team

Examples

```
## Not run:
 ## Version 6, for version 5 see the Wiki

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Get missing data patterns for each study separately
  patterns_split <- ds.mdPattern(x = "D", type = "split", datasources = connections)

  # View results for study1
  print(patterns_split$study1$pattern)
  #      var1 var2 var3
  # 150    1    1    1  0    <- 150 obs complete
  #  25    0    1    1  1    <- 25 obs missing var1
  #       25    0    0 25    <- Summary: 25 missing per variable
```

```
  # Get pooled missing data patterns across studies
  patterns_pooled <- ds.mdPattern(x = "D", type = "combine", datasources = connections)
  print(patterns_pooled$pattern)

  # Example with suppressed patterns:
  # If study1 has a pattern with count=2 (suppressed) and study2 has same pattern
  # with count=5 (valid), the pooled result will show count=5 (conservative approach)
  # A warning will indicate: "Pooled counts may underestimate the true total"

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.mean *Computes server-side vector statistical mean*

Description

This function computes the statistical mean of a given server-side vector.

Usage

```
ds.mean(
  x = NULL,
  type = "split",
  checks = FALSE,
  save.mean.Nvalid = FALSE,
  datasources = NULL
)
```

Arguments

x	a character specifying the name of a numerical vector.
type	a character string that represents the type of analysis to carry out. This can be set as 'combine', 'combined', 'combines', 'split', 'splits', 's', 'both' or 'b'. For more information see **Details**.
checks	logical. If TRUE optional checks of model components will be undertaken. Default is FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed.
save.mean.Nvalid	logical. If TRUE generated values of the mean and the number of valid (non-missing) observations will be saved on the data servers. Default FALSE. For more information see **Details**.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the R function mean.

The function can carry out 3 types of analysis depending on the argument type:
(1) If type is set to 'combine', 'combined', 'combines' or 'c', a global mean is calculated.
(2) If type is set to 'split', 'splits' or 's', the mean is calculated separately for each study.
(3) If type is set to 'both' or 'b', both sets of outputs are produced.

If the argument save.mean.Nvalid is set to TRUE study-specific means and Nvalids as well as the global equivalents across all studies combined are saved in the server-side. Once the estimated means and Nvalids are written into the server-side R environments, they can be used directly to centralize the variable of interest around its global mean or its study-specific means. Finally, the isDefined internal function checks whether the key variables have been created.

Server function called: meanDS

Value

ds.mean returns to the client-side a list including:

Mean.by.Study: estimated mean, Nmissing (number of missing observations), Nvalid (number of valid observations) and Ntotal (sum of missing and valid observations) separately for each study (if type = split or type = both).
Global.Mean: estimated mean, Nmissing, Nvalid and Ntotal across all studies combined (if type = combine or type = both).
Nstudies: number of studies being analysed.
ValidityMessage: indicates if the analysis was possible.

If save.mean.Nvalid is set as TRUE, the objects Nvalid.all.studies, Nvalid.study.specific, mean.all.studies and mean.study.specific are written to the server-side.

Author(s)

DataSHIELD Development Team

See Also

ds.quantileMean to compute quantiles.

ds.summary to generate the summary of a variable.

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Calculate the mean of a vector in the server-side

  ds.mean(x = "D$LAB_TSC",
          type = "split",
          checks = FALSE,
          save.mean.Nvalid = FALSE,
          datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.meanByClass` *Computes the mean and standard deviation across categories*

Description

This function calculates the mean and the standard deviation (SD) of a continuous variable for each class of up to 3 categorical variables.

Usage

```
ds.meanByClass(
  x = NULL,
  outvar = NULL,
  covar = NULL,
  type = "combine",
  datasources = NULL
)
```

Arguments

x	a character string specifying the name of the dataset or a text formula.
outvar	a character vector specifying the names of the continuous variables.
covar	a character vector specifying the names of up to 3 categorical variables
type	a character string that represents the type of analysis to carry out. type can be set as: 'combine' or 'split'. Default 'combine'. For more information see **Details**.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

The function splits the input dataset into subsets (one for each category) and calculates the mean and SD of the specified numeric variables. It is important to note that the process of generating the final table(s) can be time consuming particularly if the subsetting is done across more than one categorical variable and the run-time lengthens if the parameter type is set to 'split' as a table is then produced for each study. It is therefore advisable to run the function only for the studies of the user interested in but including only those studies in the parameter datasources.

Depending on the variable type can be carried out two analysis:
(1) 'combine': a pooled table of results is generated.
(2) 'split': a table of results is generated for each study.

Value

ds.meanByClass returns to the client-side a table or a list of tables that hold the length of the numeric variable(s) and their mean and standard deviation in each subgroup (subset).

Author(s)

DataSHIELD Development Team

See Also

ds.subsetByClass to subset by the classes of factor vector(s).

ds.subset to subset by complete cases (i.e. removing missing values), threshold, columns and rows.

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')
```

```
  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()


  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Calculate mean by class

  ds.meanByClass(x = "D",
                 outvar = c('LAB_HDL','LAB_TSC'),
                 covar = c('PM_BMI_CATEGORICAL'),
                 type = "combine",
                 datasources = connections)

  ds.meanByClass(x = "D$LAB_HDL~D$PM_BMI_CATEGORICAL",
                 type = "combine",
                 datasources = connections[1])#Only the frist server is used ("study1")

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.meanSdGp` *Computes the mean and standard deviation across groups defined by one factor*

Description

This function calculates the mean and SD of a continuous variable for each class of a single factor.

Usage

```
ds.meanSdGp(
  x = NULL,
  y = NULL,
```

```
  type = "both",
  do.checks = FALSE,
  datasources = NULL
)
```

Arguments

x	a character string specifying the name of a numeric continuous variable.
y	a character string specifying the name of a categorical variable of class factor.
type	a character string that represents the type of analysis to carry out. This can be set as: "combine", "split" or "both". Default "both". For more information see **Details**.
do.checks	logical. If TRUE the administrative checks are undertaken to ensure that the input objects are defined in all studies and that the variables are of equivalent class in each study. Default is FALSE to save time.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function calculates the mean, standard deviation (SD), N (number of observations) and the standard error of the mean (SEM) of a continuous variable broken down into subgroups defined by a single factor.

There are important differences between ds.meanSdGp function compared to the function ds.meanByClass:

(A) ds.meanSdGp does not actually subset the data it simply calculates the required statistics and reports them. This means you cannot use this function if you wish to physically break the data into subsets. On the other hand, it makes the function very much faster than ds.meanByClass if you do not need to create physical subsets.
(B) ds.meanByClass allows you to specify up to three categorising factors, but ds.meanSdGp only allows one. However, this is not a serious problem. If you have two factors (e.g. sex with two levels [0,1] and BMI.categorical with three levels [1,2,3]) you simply need to create a new factor that combines the two together in a way that gives each combination of levels a different value in the new factor. So, in the example given, the calculation newfactor = (3*sex) + BMI gives you six values:
(1) sex = 0 and BMI = 1 -> newfactor = 1
(2) sex = 0 and BMI = 2 -> newfactor = 2
(3) sex = 0 and BMI = 3 -> newfactor = 3
(4) sex = 1 and BMI = 1 -> newfactor = 4
(5) sex = 1 and BMI = 2 -> newfactor = 5
(6) sex = 1 and BMI = 3 -> newfactor = 6

(C) At present, ds.meanByClass calculates the sample size in each group to mean the total sample size (i.e. it includes all observations in each group regardless of whether or not they include missing values for the continuous variable or the factor). The calculation of sample size in each group by ds.meanSdGp always reports the number of observations that are non-missing both for the continuous variable and the factor. This makes sense - in the case of ds.meanByClass, the total size of the physical subsets was important, but when it comes down only to ds.meanSdGp which undertakes

analysis without physical subsetting, it is only the observations with non-missing values in both variables that contribute to the calculation of means and SDs within each group and so it is logical to consider those counts as primary. The only reference ds.meanSdGp makes to missing counts is in the reporting of Ntotal and Nmissing overall (ie not broken down by group).

For the future, we plan to extend ds.meanByClass to report both total and non-missing counts in subgroups.

Depending on the variable type can be carried out different analysis:
(1) "combine": a pooled table of results is generated.
(2) "split" a table of results is generated for each study.
(3) "both" both sets of outputs are produced.

Server function called: meanSdGpDS

Value

ds.meanSdGp returns to the client-side the mean, SD, Nvalid and SEM combined across studies and/or separately for each study, depending on the argument type.

Author(s)

DataSHIELD Development Team

See Also

ds.subsetByClass to subset by the classes of factor vector(s).

ds.subset to subset by complete cases (i.e. removing missing values), threshold, columns and rows.

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
```

```
                user = "administrator", password = "datashield_test&",
                table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")


 #Example 1: Calculate the mean, SD, Nvalid and SEM of the continuous variable age.60 (age in
 #years centralised at 60), broken down by time.id (a six level factor relating to survival time)
  #and report the pooled results combined across studies.

  ds.meanSdGp(x = "D$age.60",
              y = "D$time.id",
              type = "combine",
              do.checks = FALSE,
              datasources = connections)

 #Example 2: Calculate the mean, SD, Nvalid and SEM of the continuous variable age.60 (age in
 #years centralised at 60), broken down by time.id (a six level factor relating to survival time)
  #and report both study-specific results and the pooled results combined across studies.
  #Save the returned output to msg.b.

  ds.meanSdGp(x = "D$age.60",
              y = "D$time.id",
              type = "both",
              do.checks = FALSE,
              datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.merge *Merges two data frames in the server-side*

Description

Merges (links) two data frames together based on common values in defined vectors in each data frame.

Usage

```
ds.merge(
  x.name = NULL,
  y.name = NULL,
  by.x.names = NULL,
  by.y.names = NULL,
  all.x = FALSE,
```

```
  all.y = FALSE,
  sort = TRUE,
  suffixes = c(".x", ".y"),
  no.dups = TRUE,
  incomparables = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

x.name	a character string specifying the name of the first data frame to be merged. The length of the string should be less than the specified threshold for the nfilter.stringShort which is one of the disclosure prevention checks in DataSHIELD.
y.name	a character string specifying the name of the second data frame to be merged. The length of the string should be less than the specified threshold for the nfilter.stringShort which is one of the disclosure prevention checks in DataSHIELD.
by.x.names	a character string or a vector of names specifying of the column(s) in data frame x.name for merging.
by.y.names	a character string or a vector of names specifying of the column(s) in data frame y.name for merging.
all.x	logical. If TRUE then extra rows will be added to the output, one for each row in x.name that has no matching row in y.name. If FALSE the rows with data from both data frames are included in the output. Default FALSE.
all.y	logical. If TRUE then extra rows will be added to the output, one for each row in y.name that has no matching row in x.name. If FALSE the rows with data from both data frames are included in the output. Default FALSE.
sort	logical. If TRUE the merged result is sorted on elements in the by.x.names and by.y.names columns. Default TRUE.
suffixes	a character vector of length 2 specifying the suffixes to be used for making unique common column names in the two input data frames when they both appear in the merged data frame.
no.dups	logical. Suffixes are appended in more cases to avoid duplicated column names in the merged data frame. Default TRUE (FALSE before R version 3.5.0).
incomparables	values that cannot be matched. This is intended to be used for merging on one column, so these are incomparable values of that column. For more information see match in native R merge function.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default merge.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the native R function merge. There are some changes compared with the native R function in choosing which variables to use to merge the data frames, the function merge

is very flexible. For example, you can choose to merge using all vectors that appear in both data frames. However, for ds.merge in DataSHIELD it is required that all the vectors which dictate the merging are explicitly identified for both data frames using the by.x.names and by.y.names arguments.

Server function called: mergeDS

Value

ds.merge returns the merged data frame that is written on the server-side. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Create two data frames with a common column

  ds.dataFrame(x = c("D$LAB_TSC","D$LAB_TRIG","D$LAB_HDL","D$LAB_GLUC_ADJUSTED"),
               completeCases = TRUE,
               newobj = "df.x",
               datasources = connections)
```

```
ds.dataFrame(x = c("D$LAB_TSC","D$GENDER","D$PM_BMI_CATEGORICAL","D$PM_BMI_CONTINUOUS"),
             completeCases = TRUE,
             newobj = "df.y",
             datasources = connections)

 # Merge data frames using the common variable "LAB_TSC"

 ds.merge(x.name = "df.x",
          y.name = "df.y",
          by.x.names = "df.x$LAB_TSC",
          by.y.names = "df.y$LAB_TSC",
          all.x = TRUE,
          all.y = TRUE,
          sort = TRUE,
          suffixes = c(".x", ".y"),
          no.dups = TRUE,
          newobj = "df.merge",
          datasources = connections)

 # clear the Datashield R sessions and logout
 datashield.logout(connections)

## End(Not run)
```

`ds.message` *Returns server-side messages to the client-side*

Description

This function allows for error messages arising from the running of a server-side assign function to be returned to the client-side.

Usage

```
ds.message(message.obj.name = NULL, datasources = NULL)
```

Arguments

`message.obj.name`	is a character string specifying the name of the list that contains the message.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Errors arising from aggregate server-side functions can be returned directly to the client-side. But this is not possible for server-side assign functions because they are designed specifically to write

objects to the server-side and to return no meaningful information to the client-side. Otherwise, users may be able to use assign functions to return disclosive output to the client-side.

Server-side functions from which error messages are to be made available are designed to be able to write the designated error message to the `$serversideMessage` object into the list that is saved on the server-side as the primary output of that function. So only valid server-side functions of DataSHIELD can write a `$studysideMessage`. The error message is a string that cannot exceed a length of `nfilter.string` a default of 80 characters.

Server function called: `messageDS`

Value

`ds.message` returns a list object from each study, containing the message that has been written by DataSHIELD into `$studysideMessage`.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Use a ds.asCharacter assign function to create the message in the server-side

  ds.asCharacter(x.name = "D$LAB_TRIG",
                 newobj = "vector1",
```

```
                datasources = connections)

  #Return the message to the client-side

  ds.message(message.obj.name = "vector1",
             datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.metadata` *Gets the metadata associated with a variable held on the server*

Description

This function gets the metadata of a variable stored on the server.

Usage

```
ds.metadata(x = NULL, datasources = NULL)
```

Arguments

`x`	a character string specifying the name of the object.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Server function `metadataDS` is called examines the attributes associated with the variable which are non-disclosive.

Value

`ds.metadata` returns to the client-side the metadata of associated to an object held at the server.

Author(s)

Stuart Wheater, DataSHIELD Development Team

Examples

```
## Not run:

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Get the metadata associated with variable 'D'
  ds.metadata(x = 'D$LAB_TSC', datasources = connections)

  # clear the Datashield R sessions and logout
  DSI::datashield.logout(connections)

## End(Not run)
```

`ds.mice` *Multivariate Imputation by Chained Equations*

Description

This function calls the miceDS that is a wrapper function of the mice from the mice R package. The function creates multiple imputations (replacement values) for multivariate missing data. The method is based on Fully Conditional Specification, where each incomplete variable is imputed by a separate model. The MICE algorithm can impute mixes of continuous, binary, unordered categorical and ordered categorical data. In addition, MICE can impute continuous two-level data, and maintain consistency between imputations by means of passive imputation. It is recommended that the imputation is done in each datasource separately. Otherwise the user should make sure that the input data have the same columns in all datasources and in the same order.

Usage

```
ds.mice(
  data = NULL,
  m = 5,
  maxit = 5,
  method = NULL,
  predictorMatrix = NULL,
  post = NULL,
  seed = NA,
  newobj_mids = NULL,
  newobj_df = NULL,
  datasources = NULL
)
```

Arguments

data	a data frame or a matrix containing the incomplete data.
m	Number of multiple imputations. The default is m=5.
maxit	A scalar giving the number of iterations. The default is 5.
method	Can be either a single string, or a vector of strings with length ncol(data), specifying the imputation method to be used for each column in data. If specified as a single string, the same method will be used for all blocks. The default imputation method (when no argument is specified) depends on the measurement level of the target column, as regulated by the defaultMethod argument in native R mice function. Columns that need not be imputed have the empty method "".
predictorMatrix	A numeric matrix of ncol(data) rows and ncol(data) columns, containing 0/1 data specifying the set of predictors to be used for each target column. Each row corresponds to a variable to be imputed. A value of 1 means that the column variable is used as a predictor for the target variables (in the rows). By default, the predictorMatrix is a square matrix of ncol(data) rows and columns with all 1's, except for the diagonal.
post	A vector of strings with length ncol(data) specifying expressions as strings. Each string is parsed and executed within the sampler() function to post-process imputed values during the iterations. The default is a vector of empty strings, indicating no post-processing. Multivariate (block) imputation methods ignore the post parameter.
seed	either NA (default) or "fixed". If seed is set to "fixed" then a fixed seed random number generator which is study-specific is used.
newobj_mids	a character string that provides the name for the output mids object that is stored on the data servers. Default mids_object.
newobj_df	a character string that provides the name for the output dataframes that are stored on the data servers. Default imputationSet. For example, if m=5, and newobj_df="imputationSet", then five imputed dataframes are saved on the servers with names imputationSet.1, imputationSet.2, imputationSet.3, imputationSet.4, imputationSet.5.

datasources — a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

For additional details see the help header of mice function in native R mice package.

Value

a list with three elements: the method, the predictorMatrix and the post.

Author(s)

Demetris Avraam for DataSHIELD Development Team

---

ds.names *Return the names of a list object*

Description

Returns the names of a designated server-side list

Usage

```
ds.names(xname = NULL, datasources = NULL)
```

Arguments

xname — a character string specifying the name of the list.

datasources — a list of DSConnection-class objects obtained after login that represent the particular data sources (studies) to be addressed by the function call. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_default.

Details

ds.names calls aggregate function namesDS. This function is similar to the native R function names but it does not subsume all functionality, for example, it only works to extract names that already exist, not to create new names for objects. The function is restricted to objects of type list, but this includes objects that have a primary class other than list but which return TRUE to the native R function is.list. As an example this includes the multi-component object created by fitting a generalized linear model using ds.glmSLMA. The resultant object saved on each server separately is formally of class "glm" and "ls" but responds TRUE to is.list(),

Value

ds.names returns to the client-side the names of a list object stored on the server-side.

Author(s)

Amadou Gaye, updated by Paul Burton for DataSHIELD development team 25/06/2020

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Create a list in the server-side

  ds.asList(x.name = "D",
            newobj = "D.list",
            datasources = connections)

  #Get the names of the list

  ds.names(xname = "D.list",
           datasources = connections)


  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

ds.ns *Generate a Basis Matrix for Natural Cubic Splines*

Description

This function is based on the native R function ns from the splines package. This function generate the B-spline basis matrix for a natural cubic spline.

Usage

```
ds.ns(
  x,
  df = NULL,
  knots = NULL,
  intercept = FALSE,
  Boundary.knots = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

x	the predictor variable. Missing values are allowed.
df	degrees of freedom. One can supply df rather than knots; ns() then chooses df - 1 - intercept knots at suitably chosen quantiles of x (which will ignore missing values). The default, df = NULL, sets the number of inner knots as length(knots).
knots	breakpoints that define the spline. The default is no knots; together with the natural boundary conditions this results in a basis for linear regression on x. Typical values are the mean or median for one knot, quantiles for more knots. See also Boundary.knots.
intercept	if TRUE, an intercept is included in the basis; default is FALSE.
Boundary.knots	boundary points at which to impose the natural boundary conditions and anchor the B-spline basis (default the range of the data). If both knots and Boundary.knots are supplied, the basis parameters do not depend on x. Data can extend beyond Boundary.knots.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default ns.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

ns is native R is based on the function splineDesign. It generates a basis matrix for representing the family of piecewise-cubic splines with the specified sequence of interior knots, and the natural boundary conditions. These enforce the constraint that the function is linear beyond the boundary knots, which can either be supplied or default to the extremes of the data. A primary use is in modelling formula to directly specify a natural spline term in a model.

Value

A matrix of dimension length(x) * df where either df was supplied or if knots were supplied, df = length(knots) + 1 + intercept. Attributes are returned that correspond to the arguments to ns, and explicitly give the knots, Boundary.knots etc for use by predict.ns(). The object is assigned at each serverside.

Author(s)

Demetris Avraam for DataSHIELD Development Team

`ds.numNA` *Gets the number of missing values in a server-side vector*

Description

This function helps to know the number of missing values in a vector that is stored on the server-side.

Usage

```
ds.numNA(x = NULL, datasources = NULL)
```

Arguments

`x`	a character string specifying the name of the vector.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The number of missing entries are counted and the total for each study is returned.

Server function called: `numNaDS`

Value

`ds.numNA` returns to the client-side the number of missing values on a server-side vector.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Get the number of missing values on a server-side vector

  ds.numNA(x = "D$LAB_TSC",
           datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)



## End(Not run)
```

`ds.qlspline` *Basis for a piecewise linear spline with meaningful coefficients*

Description

This function is based on the native R function `qlspline` from the `lspline` package. This function computes the basis of piecewise-linear spline such that, depending on the argument marginal, the coefficients can be interpreted as (1) slopes of consecutive spline segments, or (2) slope change at consecutive knots.

Usage

```
ds.qlspline(
  x,
  q,
  na.rm = TRUE,
  marginal = FALSE,
  names = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

x	the name of the input numeric variable
q	numeric, a single scalar greater or equal to 2 for a number of equal-frequency intervals along x or a vector of numbers in (0; 1) specifying the quantiles explicitly.
na.rm	logical, whether NA should be removed when calculating quantiles, passed to na.rm of quantile. Default set to TRUE
marginal	logical, how to parametrise the spline, see Details
names	character, vector of names for constructed variables
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default qlspline.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

If marginal is FALSE (default) the coefficients of the spline correspond to slopes of the consecutive segments. If it is TRUE the first coefficient correspond to the slope of the first segment. The consecutive coefficients correspond to the change in slope as compared to the previous segment. Function qlspline wraps lspline and calculates the knot positions to be at quantiles of x. If q is a numerical scalar greater or equal to 2, the quantiles are computed at seq(0, 1, length.out = q + 1)[-c(1, q+1)], i.e. knots are at q-tiles of the distribution of x. Alternatively, q can be a vector of values in [0; 1] specifying the quantile probabilities directly (the vector is passed to argument probs of quantile).

Value

an object of class "lspline" and "matrix", which its name is specified by the newobj argument (or its default name "qlspline.newobj"), is assigned on the serverside.

Author(s)

Demetris Avraam for DataSHIELD Development Team

`ds.quantileMean` *Computes the quantiles of a server-side variable*

Description

This function calculates the mean and quantile values of a server-side quantitative variable.

Usage

```
ds.quantileMean(x = NULL, type = "combine", datasources = NULL)
```

Arguments

`x`	a character string specifying the name of the numeric vector.
`type`	a character that represents the type of graph to display. This can be set as `'combine'` or `'split'`. For more information see **Details**.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function does not return the minimum and maximum values because they are potentially disclosive.

Depending on the argument `type` can be carried out two types of analysis:
(1) `type = 'combine'` pooled values are displayed
(2) `type = 'split'` summaries are returned for each study.

Server functions called: `quantileMeanDS`, `length` and `numNaDS`

Value

`ds.quantileMean` returns to the client-side the quantiles and statistical mean of a server-side numeric vector.

Author(s)

DataSHIELD Development Team

See Also

`ds.mean` to compute the statistical mean.

`ds.summary` to generate the summary of a variable.

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Get the quantiles and mean of a server-side variable

  ds.quantileMean(x = "D$LAB_TRIG",
                  type = "combine",
                  datasources = connections)


  # clear the Datashield R sessions and logout
  datashield.logout(connections)



## End(Not run)
```

---

ds.ranksSecure *Secure ranking of a vector across all sources*

Description

Securely generate the ranks of a numeric vector and estimate true global quantiles across all data sources simultaneously

Usage

```
ds.ranksSecure(
  input.var.name = NULL,
  quantiles.for.estimation = "0.05-0.95",
  generate.quantiles = TRUE,
  output.ranks.df = NULL,
  summary.output.ranks.df = NULL,
  ranks.sort.by = "ID.orig",
  shared.seed.value = 10,
  synth.real.ratio = 2,
  NA.manage = "NA.delete",
  rm.residual.objects = TRUE,
  monitor.progress = FALSE,
  datasources = NULL
)
```

Arguments

`input.var.name` a character string in a format that can pass through the DataSHIELD R parser which specifies the name of the vector to be ranked. Needs to have same name in each data source.

`quantiles.for.estimation`
one of a restricted set of character strings. To mitigate disclosure risk only the following set of quantiles can be generated: c(0.025,0.05,0.10,0.20,0.25,0.30,0.3333,0.40,0.50,0.60,0.666 0.70,0.75,0.80,0.90,0.95,0.975). The allowable formats for the argument are of the general form: "0.025-0.975" where the first number is the lowest quantile to be estimated and the second number is the equivalent highest quantile to estimate. These two quantiles are then estimated along with all allowable quantiles in between. The allowable argument values are then: "0.025-0.975", "0.05-0.95", "0.10-0.90", "0.20-0.80". Two alternative values are "quartiles" i.e. c(0.25,0.50,0.75), and "median" i.e. c(0.50). The default value is "0.05-0.95". If the sample size is so small that an extreme quartile could be disclosive the function will be terminated and an error message returned telling you that you might try using an argument with a narrower set of quantiles. This disclosure trap will be triggered if the total number of subjects across all studies divided by the total number of quantile values being estimated is less than or equal to nfilter.tab (the minimum cell size in a contingency table).

`generate.quantiles`
a logical value indicating whether the ds.ranksSecure function should carry on to estimate the key quantile values specified by argument <quantiles.for.estimation> or should stop once the global ranks have been created and written to the server-side. Default is TRUE and as the key quantiles are generally non-disclosive this is usually the setting to use. But, if there is some abnormal configuration of the clusters of values that are being ranked such that some values are treated as being missing and the processing stops, then setting generate.quantiles to FALSE allows the generation of ranks to complete so they can then be used for non-parametric analysis, even if the key values cannot be estimated. A real example of an unusual configuration was in a reasonably large dataset of survival times,

where a substantial proportion of survival profiles were censored at precisely 10 years. This meant that the 97.5 the former was allocated the value NA. This stopped processing of the ranks which could then be enabled by setting generate.quantiles to FALSE. However, if this problem is detected an error message is returned which indicates that in some cases (as in this case in fact) the problem can be circumvented by selecting a narrow range of key quantiles to estimate. In this case, in fact, this simply required changing the <quantiles.for.estimation> argument from "0.025-0.975" to "0.05-0.95".

`output.ranks.df`

a character string in a format that can pass through the DataSHIELD R parser which specifies an optional name for the data.frame written to the serverside on each data source that contains 11 of the key output variables from the ranking procedure pertaining to that particular data source. This includes the global ranks and quantiles of each value of the V2BR (i.e. the values are ranked across all studies simultaneously). If no name is specified, the default name is allocated as "full.ranks.df". This data.frame contains disclosive information and cannot therefore be passed to the clientside.

`summary.output.ranks.df`

a character string in a format that can pass through the DataSHIELD R parser which specifies an optional name for the summary data.frame written to the serverside on each data source that contains 5 of the key output variables from the ranking procedure pertaining to that particular data source. This again includes the global ranks and quantiles of each value of the V2BR (i.e. the values are ranked across all studies simultaneously). If no name is specified, the default name is allocated as "summary.ranks.df" This data.frame contains disclosive information and cannot therefore be passed to the clientside.

`ranks.sort.by` a character string taking two possible values. These are "ID.orig" and "vals.orig". These define the order in which the output.ranks.df and summary.output.ranks.df data frames are presented. If the argument is set as "ID.orig" the order of rows in the output data frames are precisely the same as the order of original input vector that is being ranked (i.e. V2BR). This means the ranks can simply be cbinded to the matrix, data frame or tibble that originally included V2BR so it also includes the corresponding ranks. If it is set as "vals.orig" the output data frames are in order of increasing magnitude of the original values of V2BR. Default value is "ID.orig".

`shared.seed.value`

an integer value which is used to set the random seed generator in each study. Initially, the seed is set to be the same in all studies, so the order and parameters of the repeated encryption procedures are precisely the same in each study. Then a study-specific modification of the seed in each study ensures that the procedures initially generating the masking pseudodata (which are then subject to the same encryption procedures as the real data) are different in each study. For further information about the shared seed and how we intend to transmit it in the future, please see the detailed associated header document.

`synth.real.ratio`

an integer value specifying the ratio between the number of masking pseudodata values generated in each study compared to the number of real data values in V2BR.

`NA.manage` character string taking three possible values: "NA.delete", "NA.low","NA.hi". This argument determines how missing values are managed before ranking. "NA.delete" results in all missing values being removed prior to ranking. This means that the vector of ranks in each study is shorter than the original vector of V2BR values by an amount corresponding to the number of missing values in V2BR in that study. Any rows containing missing values in V2BR are simply removed before the ranking procedure is initiated so the order of rows without missing data is unaltered. "NA.low" indicates that all missing values should be converted to a new value that has a meaningful magnitude that is lower (more negative or less positive) than the lowest non-missing value of V2BR in any of the studies. This means, for example, that if there are a total of M values of V2BR that are missing across all studies, there will be a total of M observations that are ranked lowest each with a rank of (M+1)/2. So if 7 are missing the lowest 7 ranks will be 4,4,4,4,4,4,4 and if 4 are missing the first 4 ranks will be 2.5,2.5,2.5,2.5. "NA.hi" indicates that all missing values should be converted to a new value that has a meaningful magnitude that is higher(less negative or more positive)than the highest non-missing value of V2BR in any of the studies. This means, for example, that if there are a total of M values of V2BR that are missing across all studies and N non-missing values, there will be a total of M observations that are ranked highest each with a rank of (2N-M+1)/2. So if there are a total of 1000 V2BR values and 9 are missing the highest 9 ranks will be 996, 996 ... 996. If NA.manage is either "NA.low" or "NA.hi" the final rank vector in each study will have the same length as the V2BR vector in that same study. 2.5,2.5,2.5,2.5. The default value of the "NA.manage" argument is "NA.delete"

`rm.residual.objects` logical value. Default = TRUE: at the beginning and end of each run of ds.ranksSecure delete all extraneous objects that are otherwise left behind. These are not usually needed, but could be of value if one were investigating a problem with the ranking. FALSE: do not delete the residual objects

`monitor.progress` logical value. Default = FALSE. If TRUE, function outputs information about its progress.

`datasources` specifies the particular opal object(s) to use. If the <datasources> argument is not specified (NULL) the default set of opals will be used. If <datasources> is specified, it should be set without inverted commas: e.g. datasources=opals.em. If you wish to apply the function solely to e.g. the second opal server in a set of three, the argument can be specified as: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set you specify: e.g. datasources=opals.em[c(1,3)].

Details

ds.ranksSecure is a clientside function which calls a series of other clientside and serverside functions to securely generate the global ranks of a numeric vector "V2BR" (vector to be ranked) in order to set up analyses on V2BR based on non-parametric methods, some types of survival analysis and to derive true global quantiles (such as the median, lower (25 and the 95 global quantiles are, in general, different to the mean or median of the equivalent quantiles calculated independently

in each data source separately. For more details about the cluster of functions that collectively enable secure global ranking and estimation of global quantiles see the associated document entitled "secure.global.ranking.docx".

Value

the data frame objects specified by the arguments output.ranks.df and summary.output.ranks.df. These are written to the serverside in each study. Provided the sort order is consistent these data frames can be cbinded to any other data frame, matrix or tibble object containing V2BR or to the V2BR vector itself, allowing the global ranks and quantiles to be analysed rather than the actual values of V2BR. The last call within the ds.ranksSecure function is to another clientside function ds.extractQuantile (for further details see header for that function). This returns an additional data frame "final.quantile.df" of which the first column is the vector of key quantiles to be estimated as specified by the argument <quantiles.for.estimation> and the second column is the list of precise values of V2BR which correspond to these key quantiles. Because the serverside functions associated with ds.ranksSecure and ds.extractQuantile block potentially disclosive output (see information for parameter quantiles.for.estimation) the "final.quantile.df" is returned to the client allowing the direct reporting of V2BR values corresponding to key quantiles such as the quartiles, the median and 95th percentile etc. In addition a copy of the same data frame is also written to the serverside in each study allowing the value of key quantiles such as the median to be incorporated directly in calculations or transformations on the serverside regardless in which study (or studies) those key quantile values have occurred.

Author(s)

Paul Burton 4th November, 2021

`ds.rbind` *Combines R objects by rows in the server-side*

Description

It takes a sequence of vector, matrix or data-frame arguments and combines them by rows to produce a matrix.

Usage

```
ds.rbind(
  x = NULL,
  DataSHIELD.checks = FALSE,
  force.colnames = NULL,
  newobj = NULL,
  datasources = NULL,
  notify.of.progress = FALSE
)
```

Arguments

x	a character vector with the name of the objects to be combined.
DataSHIELD.checks	logical, if TRUE checks that all input objects exist and are of an appropriate class.
force.colnames	can be NULL or a vector of characters that specifies column names of the output object.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Defaults rbind.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa
notify.of.progress	specifies if console output should be produced to indicate progress. Default FALSE.

Details

A sequence of vector, matrix or data-frame arguments is combined by rows to produce a matrix on the server-side.

In DataSHIELD.checks the checks are relatively slow. Default DataSHIELD.checks value is FALSE.

If force.colnames is NULL column names are inferred from the names or column names of the first object specified in the x argument. The vector of column names must have the same number of elements as the columns in the output object.

Server functions called: rbindDS.

Value

ds.rbind returns a matrix combining the rows of the R objects specified in the function which is written to the server-side. It also returns two messages to the client-side with the name of newobj that has been created in each data source and DataSHIELD.checks result.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
```

```
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Combining R objects by rows


  ds.rbind(x = "D", #data frames in the server-side to be conbined
                    #(see above the connection to the Opal servers)
           DataSHIELD.checks = FALSE,
           force.colnames = NULL,
         newobj = "D.rbind", # name for the output object that is stored in the data servers
           datasources = connections, # All Opal servers are used
                                      #(see above the connection to the Opal servers)
           notify.of.progress = FALSE)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.rBinom` *Generates Binomial distribution in the server-side*

Description

Generates random (pseudorandom) non-negative integers from a Binomial distribution. Also, `ds.rBinom` allows creating different vector lengths in each server.

Usage

```
ds.rBinom(
  samp.size = 1,
  size = 0,
  prob = 1,
  newobj = NULL,
```

```
  seed.as.integer = NULL,
  return.full.seed.as.set = FALSE,
  datasources = NULL
)
```

Arguments

samp.size	an integer value or an integer vector that defines the length of the random numeric vector to be created in each source.
size	a positive integer that specifies the number of Bernoulli trials.
prob	a numeric scalar value or vector in range 0 > prob > 1 which specifies the probability of a positive response (i.e. 1 rather than 0).
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default rbinom.newobj.
seed.as.integer	an integer or a NULL value which provides the random seed in each data source.
return.full.seed.as.set	logical, if TRUE will return the full random number seed in each data source (a numeric vector of length 626). If FALSE it will only return the trigger seed value you have provided. Default is FALSE.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Creates a vector of random or pseudorandom non-negative integer values distributed with a Binomial distribution. The ds.rBinom function's arguments specify the number of trials, the success probability, the length and the seed of the output vector in each source.

To specify a different size in each source, you can use a character vector (..., size="vector.of.sizes"...) or the datasources parameter to create the random vector for one source at a time, changing size as required. The default value for size = 1 which simulates binary outcomes (all observations 0 or 1).

To specify different prob in each source, you can use an integer or character vector (..., prob="vector.of.probs"...) or the datasources parameter to create the random vector for one source at a time, changing prob as required.

If seed.as.integer is an integer e.g. 5 and there is more than one source (N) the seed is set as 5*N. For example, in the first study the seed is set as 938*1, in the second as 938*2 up to 938*N in the Nth study.

If seed.as.integer is set as 0 all sources will start with the seed value 0 and all the random number generators will, therefore, start from the same position. Besides, to use the same starting seed in all studies but do not wish it to be 0, you can use datasources argument to generate the random number vectors one source at a time.

Server functions called: rBinomDS and setSeedDS.

Value

ds.rBinom returns random number vectors with a Binomial distribution for each study, taking into account the values specified in each parameter of the function. The output vector is written to the server-side. If requested, it also returned to the client-side the full 626 lengths random seed vector generated in each source (see info for the argument return.full.seed.as.set).

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")

  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Generating the vectors in the Opal servers
 ds.rBinom(samp.size=c(13,20,25), #the length of the vector created in each source is different
  size=as.character(c(10,23,5)),   #Bernoulli trials change in each source
  prob=c(0.6,0.1,0.5), #Probability  changes in each source
  newobj="Binom.dist",
  seed.as.integer=45,
  return.full.seed.as.set=FALSE,
  datasources=connections)   #all the Opal servers are used, in this case 3
                             #(see above the connection to the servers)

  ds.rBinom(samp.size=15,
            size=4,
            prob=0.7,
```

```
            newobj="Binom.dist",
            seed.as.integer=324,
            return.full.seed.as.set=FALSE,
            datasources=connections[2]) #only the second  Opal server is used ("study2")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.recodeLevels` *Recodes the levels of a server-side factor vector*

Description

The function replaces the levels of a factor by the specified new ones.

Usage

```
ds.recodeLevels(
  x = NULL,
  newCategories = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`x`	a character string specifying the name of a factor variable.
`newCategories`	a character vector specifying the new levels. Its length must be equal or greater to the current number of levels.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `recodelevels.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function is similar to native R function `levels()`.

It can for example be used to merge two classes into one, to add a level(s) to a vector or to rename (i.e. re-label) the levels of a vector.

Server function called: `levels()`

Value

`ds.recodeLevels` returns to the server-side a variable of type factor with the replaces levels.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Recode the levels of a factor variable

  ds.recodeLevels(x = "D$PM_BMI_CATEGORICAL",
                  newCategories = c("1","2","3"),
                  newobj = "BMI_CAT",
                  datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.recodeValues *Recodes server-side variable values*

Description

This function takes specified values of elements in a vector and converts them to a matched set of alternative specified values.

Usage

```
ds.recodeValues(
  var.name = NULL,
  values2replace.vector = NULL,
  new.values.vector = NULL,
  missing = NULL,
  newobj = NULL,
  datasources = NULL,
  notify.of.progress = FALSE
)
```

Arguments

var.name	a character string providing the name of the variable to be recoded.
values2replace.vector	a numeric or character vector specifying the values in the variable var.name to be replaced.
new.values.vector	a numeric or character vector specifying the new values.
missing	If supplied, any missing values in var.name will be replaced by this value. Must be of length 1. If the analyst want to recode only missing values then it should also specify an identical vector of values in both arguments values2replace.vector and new.values.vector. Otherwise please look the ds.replaceNA function.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default recodevalues.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa
notify.of.progress	logical. If TRUE console output should be produced to indicate progress. Default FALSE.

Details

This function recodes individual values with new individual values. This can apply to numeric and character values, factor levels and NAs. One particular use of ds.recodeValues is to convert NAs to an explicit value. This value is specified in the argument missing. If the user want to recode only missing values, then it should also specify an identical vector of values in both arguments values2replace.vector and new.values.vector (see Example 2 below). Server function called: recodeValuesDS

Value

Assigns to each server a new variable with the recoded values. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: recode the levels of D$GENDER
  ds.recodeValues(var.name = "D$GENDER",
                  values2replace.vector = c(0,1),
                  new.values.vector = c(10,20),
                  newobj = 'gender_recoded',
                  datasources = connections)

  # Example 2: recode NAs in D$PM_BMI_CATEGORICAL
  ds.recodeValues(var.name = "D$PM_BMI_CATEGORICAL",
                  values2replace.vector = c(1,2),
                  new.values.vector = c(1,2),
                  missing = 99,
                  newobj = 'bmi_recoded',
                  datasources = connections)
```

```
  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.rep` *Creates a repetitive sequence in the server-side*

Description

Creates a repetitive sequence by repeating the specified scalar number, vector or list in each data source.

Usage

```
ds.rep(
  x1 = NULL,
  times = NA,
  length.out = NA,
  each = 1,
  source.x1 = "clientside",
  source.times = NULL,
  source.length.out = NULL,
  source.each = NULL,
  x1.includes.characters = FALSE,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`x1`	an scalar number, vector or list.
`times`	an integer from clientside or a serverside integer or vector.
`length.out`	a clientside integer or a serverside integer or vector.
`each`	a clientside or serverside integer.
`source.x1`	the source `x1` argument. It can be "clientside" or "c" and serverside or "s".
`source.times`	see `source.x1`
`source.length.out`	see `source.x1`
`source.each`	see `source.x1`
`x1.includes.characters`	Boolean parameter which specifies if the `x1` is a character.
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `seq.vect`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

All arguments that can denote in a clientside or a serverside (i.e. x1, times, length.out or each).

Server function called: repDS.

Value

ds.rep returns in the server-side a vector with the specified repetitive sequence. Also, two validity messages are returned to the client-side the name of newobj that has been created in each data source and if it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

    require('DSI')
    require('DSOpal')
    require('dsBaseClient')

    builder <- DSI::newDSLoginBuilder()
    builder$append(server = "study1",
                   url = "http://192.168.56.100:8080/",
                   user = "administrator", password = "datashield_test&",
                   table = "CNSIM.CNSIM1", driver = "OpalDriver")
    builder$append(server = "study2",
                   url = "http://192.168.56.100:8080/",
                   user = "administrator", password = "datashield_test&",
                   table = "CNSIM.CNSIM2", driver = "OpalDriver")
    builder$append(server = "study3",
                   url = "http://192.168.56.100:8080/",
                   user = "administrator", password = "datashield_test&",
                   table = "CNSIM.CNSIM3", driver = "OpalDriver")
    logindata <- builder$build()

  # Log onto the remote Opal training servers
    connections <- DSI::datashield.login(logins = logindata,
                                         assign = TRUE,
                                         symbol = "D")

  # Creating a repetitive sequence

     ds.rep(x1 = 4,
            times = 6,
            length.out = NA,
            each = 1,
```

```
            source.x1 = "clientside",
            source.times = "c",
            source.length.out = NULL,
            source.each = "c",
            x1.includes.characters = FALSE,
            newobj = "rep.seq",
            datasources = connections)

    ds.rep(x1 = "lung",
            times = 6,
            length.out = 7,
            each = 1,
            source.x1 = "clientside",
            source.times = "c",
            source.length.out = "c",
            source.each = "c",
            x1.includes.characters = TRUE,
            newobj = "rep.seq",
            datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.replaceNA` *Replaces the missing values in a server-side vector*

Description

This function identifies missing values and replaces them by a value or values specified by the analyst.

Usage

```
ds.replaceNA(x = NULL, forNA = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x`	a character string specifying the name of the vector.
`forNA`	a list or a vector that contains the replacement value(s), for each study. The length of the list or vector must be equal to the number of servers (studies).
`newobj`	a character string that provides the name for the output object that is stored on the data servers. Default `replacena.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function is used when the analyst prefers or requires complete vectors. It is then possible the specify one value for each missing value by first returning the number of missing values using the function ds.numNA but in most cases, it might be more sensible to replace all missing values by one specific value e.g. replace all missing values in a vector by the mean or median value. Once the missing values have been replaced a new vector is created.

Note: If the vector is within a table structure such as a data frame the new vector is appended to table structure so that the table holds both the vector with and without missing values.

Server function called: replaceNaDS

Value

ds.replaceNA returns to the server-side a new vector or table structure with the missing values replaced by the specified values. The class of the vector is the same as the initial vector.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

    require('DSI')
    require('DSOpal')
    require('dsBaseClient')

    builder <- DSI::newDSLoginBuilder()
    builder$append(server = "study1",
                   url = "http://192.168.56.100:8080/",
                   user = "administrator", password = "datashield_test&",
                   table = "CNSIM.CNSIM1", driver = "OpalDriver")
    builder$append(server = "study2",
                   url = "http://192.168.56.100:8080/",
                   user = "administrator", password = "datashield_test&",
                   table = "CNSIM.CNSIM2", driver = "OpalDriver")
    builder$append(server = "study3",
                   url = "http://192.168.56.100:8080/",
                   user = "administrator", password = "datashield_test&",
                   table = "CNSIM.CNSIM3", driver = "OpalDriver")
    logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Replace missing values in variable 'LAB_HDL' by the mean value
  # in each study
```

```
  # Get the mean value of  'LAB_HDL' for each study
  mean <- ds.mean(x = "D$LAB_HDL",
                  type = "split",
                  datasources = connections)

  # Replace the missing values using the mean for each study
  ds.replaceNA(x = "D$LAB_HDL",
               forNA = list(mean[[1]][1], mean[[1]][2], mean[[1]][3]),
               newobj = "HDL.noNA",
               datasources = connections)

  # Example 2: Replace missing values in categorical variable 'PM_BMI_CATEGORICAL'
  # with 999s

  # First check how many NAs there are in 'PM_BMI_CATEGORICAL' in each study
  ds.table(rvar = "D$PM_BMI_CATEGORICAL",
          useNA = "always")

  # Replace the missing values with 999s
  ds.replaceNA(x = "D$PM_BMI_CATEGORICAL",
               forNA = c(999,999,999),
               newobj = "bmi999")

  # Check if the NAs have been replaced correctly
  ds.table(rvar = "bmi999",
          useNA = "always")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.reShape *Reshapes server-side grouped data*

Description

Reshapes a data frame containing longitudinal or otherwise grouped data from 'wide' to 'long' format or vice-versa.

Usage

```
ds.reShape(
  data.name = NULL,
  varying = NULL,
  v.names = NULL,
  timevar.name = "time",
  idvar.name = "id",
```

```
  drop = NULL,
  direction = NULL,
  sep = ".",
  newobj = "newObject",
  datasources = NULL
)
```

Arguments

data.name	a character string specifying the name of the data frame to be reshaped.
varying	names of sets of variables in the wide format that correspond to single variables in 'long' format.
v.names	the names of variables in the 'long' format that correspond to multiple variables in the 'wide' format.
timevar.name	the variable in 'long' format that differentiates multiple records from the same group or individual. If more than one record matches, the first will be taken.
idvar.name	names of one or more variables in 'long' format that identify multiple records from the same group/individual. These variables may also be present in 'wide' format.
drop	a vector of names of variables to drop before reshaping. This can simplify the resultant output.
direction	a character string that partially matched to either 'wide' to reshape from 'long' to 'wide' format, or 'long' to reshape from 'wide' to 'long' format.
sep	a character vector of length 1, indicating a separating character in the variable names in the 'wide' format. This is used for creating good v.names and times arguments based on the names in the varying argument. This is also used to create variable names when reshaping to 'wide' format.
newobj	a character string that provides the name for the output object that is stored on the data servers. Default reshape.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is based on the native R function reshape. It reshapes a data frame containing longitudinal or otherwise grouped data between 'wide' format with repeated measurements in separate columns of the same record and 'long' format with the repeated measurements in separate records. The reshaping can be in either direction. Server function called: reShapeDS

Value

ds.reShape returns to the server-side a reshaped data frame converted from 'long' to 'wide' format or from 'wide' to long' format. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "SURVIVAL.EXPAND_NO_MISSING3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Reshape server-side grouped data

  ds.reShape(data.name = "D",
             v.names = "age.60",
             timevar.name = "time.id",
             idvar.name = "id",
             direction = "wide",
             newobj = "reshape1_obj",
             datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.rm` *Deletes server-side R objects*

Description

deletes R objects on the server-side

Usage

```
ds.rm(x.names = NULL, datasources = NULL)
```

Arguments

x.names	a character string specifying the objects to be deleted.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the native R function rm().

The fact that it is an aggregate function may be surprising because it modifies an object on the server-side, and would, therefore, be expected to be an assign function. However, as an assign function the last step in running it would be to write the modified object as newobj. But this would fail because the effect of the function is to delete the object and so it would be impossible to write it anywhere. Please note that although this calls an aggregate function there is no type argument.

Server function called: rmDS

Value

The ds.rm function deletes from the server-side the specified object. If this is successful the message "Object(s) '<x.names>' was deleted." is returned to the client-side.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
```

```
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Create an object in the server-side

  ds.assign(toAssign = "D$LAB_TSC",
            newobj = "labtsc",
            datasources = connections)

  #Delete "labtsc" object from the server-side

  ds.rm(x.names = "labtsc",
        datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.rNorm` *Generates Normal distribution in the server-side*

Description

Generates normally distributed random (pseudorandom) scalar numbers. Besides, `ds.rNorm` allows creating different vector lengths in each server.

Usage

```
ds.rNorm(
  samp.size = 1,
  mean = 0,
  sd = 1,
  newobj = "newObject",
  seed.as.integer = NULL,
  return.full.seed.as.set = FALSE,
  force.output.to.k.decimal.places = 9,
  datasources = NULL
)
```

Arguments

`samp.size`	an integer value or an integer vector that defines the length of the random numeric vector to be created in each source.
`mean`	the mean value or vector of the Normal distribution to be created.

`sd` the standard deviation of the Normal distribution to be created.

`newobj` a character string that provides the name for the output variable that is stored on the data servers. Default `newObject`.

`seed.as.integer` an integer or a NULL value which provides the random seed in each data source.

`return.full.seed.as.set` logical, if TRUE will returns the full random number seed in each data source (a numeric vector of length 626). If FALSE it will only return the trigger seed value you have provided. Default is FALSE.

`force.output.to.k.decimal.places` an integer vector that forces the output random numbers vector to have k decimals.

`datasources` a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

Creates a vector of pseudorandom numbers distributed with a Normal distribution in each data source. The `ds.rNorm` function's arguments specify the mean and the standard deviation (`sd`) of the normal distribution and the length and the seed of the output vector in each source.

To specify a different `mean` value in each source, you can use a character vector `(..., mean="vector.of.means"...)` or the `datasources` parameter to create the random vector for one source at a time, changing the `mean` as required. Default value for `mean = 0`.

To specify different `sd` value in each source, you can use a character vector `(..., sd="vector.of.sds"...` or the `datasources` parameter to create the random vector for one source at a time, changing the <mean> as required. Default value for `sd = 0`.

If `seed.as.integer` is an integer e.g. 5 and there is more than one source (N) the seed is set as 5*N. For example, in the first study the seed is set as 938*1, in the second as 938*2 up to 938*N in the Nth study.

If `seed.as.integer` is set as 0 all sources will start with the seed value 0 and all the random number generators will, therefore, start from the same position. Also, to use the same starting seed in all studies but do not wish it to be 0, you can use `datasources` argument to generate the random number vectors one source at a time.

In `force.output.to.k.decimal.places` the range of k is 1-8 decimals. If `k = 0` the output random numbers are forced to integer. If `k = 9`, no rounding of output numbers occurs. The default value of `force.output.to.k.decimal.places = 9`.

Server functions called: `rNormDS` and `setSeedDS`.

Value

`ds.rNorm` returns random number vectors with a normal distribution for each study, taking into account the values specified in each parameter of the function. The output vector is written to the server-side. If requested, it also returned to the client-side the full 626 lengths random seed vector generated in each source (see info for the argument `return.full.seed.as.set`).

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Generating the vectors in the Opal servers

 ds.rNorm(samp.size=c(10,20,45), #the length of the vector created in each source is different
        mean=c(1,6,4),       #the mean of the Normal distribution changes in each server
       sd=as.character(c(1,4,3)), #the sd of the Normal distribution changes in each server
          newobj="Norm.dist",
          seed.as.integer=2345,
          return.full.seed.as.set=FALSE,
         force.output.to.k.decimal.places=c(4,5,6), #output random numbers have different
                                                    #decimal quantity in each source
          datasources=connections) #all the Opal servers are used, in this case 3
                                   #(see above the connection to the servers)

  ds.rNorm(samp.size=10,
           mean=1.4,
           sd=0.2,
           newobj="Norm.dist",
           seed.as.integer=2345,
           return.full.seed.as.set=FALSE,
           force.output.to.k.decimal.places=1,
```

```
            datasources=connections[2]) #only the second  Opal server is used ("study2")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.rowColCalc` *Computes rows and columns sums and means in the server-side*

Description

Computes sums and means of rows or columns of a numeric matrix or data frame on the server-side.

Usage

```
ds.rowColCalc(x = NULL, operation = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x`	a character string specifying the name of a matrix or a data frame.
`operation`	a character string that indicates the operation to carry out: `"rowSums"`, `"colSums"`, `"rowMeans"` or `"colMeans"`.
`newobj`	a character string that provides the name for the output variable that is stored on the data servers. Default `rowcolcalc.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The function is similar to R base functions `rowSums`, `colSums`, `rowMeans` and `colMeans` with some restrictions.

The results of the calculation are not returned to the user if they are potentially revealing i.e. if the number of rows is less than the allowed number of observations.

Server functions called: `classDS`, `dimDS` and `colnamesDS`

Value

`ds.rowColCalc` returns to the server-side rows and columns sums and means.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()
  myvar <- list("LAB_TSC","LAB_HDL")

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE,
  variables = myvar, symbol = "D")


  #Calculate the colSums

  ds.rowColCalc(x = "D",
                operation = "colSums",
                newobj = "D.rowSums",
                datasources = connections)

  #Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.rPois` *Generates Poisson distribution in the server-side*

Description

Generates random (pseudorandom) non-negative integers with a Poisson distribution. Besides, `ds.rPois` allows creating different vector lengths in each server.

Usage

```
ds.rPois(
  samp.size = 1,
  lambda = 1,
  newobj = "newObject",
  seed.as.integer = NULL,
  return.full.seed.as.set = FALSE,
  datasources = NULL
)
```

Arguments

samp.size	an integer value or an integer vector that defines the length of the random numeric vector to be created in each source.
lambda	the number of events mean per interval.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default newObject.
seed.as.integer	an integer or a NULL value which provides the random seed in each data source.
return.full.seed.as.set	logical, if TRUE will return the full random number seed in each data source (a numeric vector of length 626). If FALSE it will only return the trigger seed value you have provided. Default is FALSE.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Creates a vector of random or pseudorandom non-negative integer values distributed with a Poisson distribution in each data source. The ds.rPois function's arguments specify lambda, the length and the seed of the output vector in each source.

To specify different lambda value in each source, you can use a character vector (..., lambda = "vector.of.lambdas"...) or the datasources parameter to create the random vector for one source at a time, changing lambda as required. Default value for lambda> = 1.

If seed.as.integer is an integer e.g. 5 and there is more than one source (N) the seed is set as 5*N. For example, in the first study the seed is set as 938*1, in the second as 938*2 up to 938*N in the Nth study.

If seed.as.integer is set as 0 all sources will start with the seed value 0 and all the random number generators will, therefore, start from the same position. Also, to use the same starting seed in all studies but do not wish it to be 0, you can use datasources argument to generate the random number vectors one source at a time.

Server functions called: rPoisDS and setSeedDS.

Value

ds.rPois returns random number vectors with a Poisson distribution for each study, taking into account the values specified in each parameter of the function. The created vectors are stored in the server-side. If requested, it also returned to the client-side the full 626 lengths random seed vector generated in each source (see info for the argument return.full.seed.as.set).

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()
  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Generating the vectors in the Opal servers
 ds.rPois(samp.size=c(13,20,25), #the length of the vector created in each source is different
       lambda=as.character(c(2,3,4)), #different mean per interval (2,3,4) in each source
         newobj="Pois.dist",
         seed.as.integer=1234,
         return.full.seed.as.set=FALSE,
         datasources=connections)  #all the Opal servers are used, in this case 3
                                   #(see above the connection to the servers)
 ds.rPois(samp.size=13,
         lambda=5,
         newobj="Pois.dist",
         seed.as.integer=1234,
         return.full.seed.as.set=FALSE,
         datasources=connections[1])  #only the first Opal server is used ("study1")
```

```
  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.rUnif *Generates Uniform distribution in the server-side*

Description

Generates uniformly distributed random (pseudorandom) scalar numbers. Besides, ds.rUnif allows creating different vector lengths in each server.

Usage

```
ds.rUnif(
  samp.size = 1,
  min = 0,
  max = 1,
  newobj = "newObject",
  seed.as.integer = NULL,
  return.full.seed.as.set = FALSE,
  force.output.to.k.decimal.places = 9,
  datasources = NULL
)
```

Arguments

samp.size	an integer value or an integer vector that defines the length of the random numeric vector to be created in each source.
min	a numeric scalar that specifies the minimum value of the random numbers in the distribution.
max	a numeric scalar that specifies the maximum value of the random numbers in the distribution.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default newObject.
seed.as.integer	an integer or a NULL value which provides the random seed in each data source.
return.full.seed.as.set	logical, if TRUE will return the full random number seed in each data source (a numeric vector of length 626). If FALSE it will only return the trigger seed value you have provided. Default is FALSE.
force.output.to.k.decimal.places	an integer or an integer vector that forces the output random numbers vector to have k decimals.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

It creates a vector of pseudorandom numbers distributed with a uniform probability in each data source. The ds.Unif function's arguments specify the minimum and maximum of the uniform distribution and the length and the seed of the output vector in each source.

To specify different min values in each source, you can use a character vector (..., min="vector.of.mins"...) or the datasources parameter to create the random vector for one source at a time, changing the min value as required. Default value for min = 0.

To specify different max values in each source, you can use a character vector (..., max="vector.of.maxs"...) or the datasources parameter to create the random vector for one source at a time, changing the max value as required. Default value for max = 1.

If seed.as.integer is an integer e.g. 5 and there is more than one source (N) the seed is set as 5*N. For example, in the first study the seed is set as 938*1, in the second as 938*2 up to 938*N in the Nth study.

If seed.as.integer is set as 0 all sources will start with the seed value 0 and all the random number generators will, therefore, start from the same position. Also, to use the same starting seed in all studies but do not wish it to be 0, you can use datasources argument to generate the random number vectors one source at a time.

In force.output.to.k.decimal.places the range of k is 1-8 decimals. If k = 0 the output random numbers are forced to an integer. If k = 9, no rounding of output numbers occurs. The default value of force.output.to.k.decimal.places = 9. If you wish to generate integers with equal probabilities in the range 1-10 you should specify min = 0.5 and max = 10.5. Default value for k = 9.

Server functions called: rUnifDS and setSeedDS.

Value

ds.Unif returns random number vectors with a uniform distribution for each study, taking into account the values specified in each parameter of the function. The created vectors are stored in the server-side. If requested, it also returned to the client-side the full 626 lengths random seed vector generated in each source (see info for the argument return.full.seed.as.set).

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
```

```
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Generating the vectors in the Opal servers

 ds.rUnif(samp.size = c(12,20,4), #the length of the vector created in each source is different
       min = as.character(c(0,2,5)), #different minumum value of the function in each source
       max = as.character(c(2,5,9)), #different maximum value of the function in each source
           newobj = "Unif.dist",
           seed.as.integer = 234,
           return.full.seed.as.set = FALSE,
           force.output.to.k.decimal.places = c(1,2,3),
           datasources = connections)   #all the Opal servers are used, in this case 3
                                        #(see above the connection to the servers)

  ds.rUnif(samp.size = 12,
           min = 0,
           max = 2,
           newobj = "Unif.dist",
           seed.as.integer = 12345,
           return.full.seed.as.set = FALSE,
           force.output.to.k.decimal.places = 2,
          datasources = connections[2]) #only the second  Opal server is used ("study2")

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.sample` *Performs random sampling and permuting of vectors, dataframes and matrices*

Description

draws a pseudorandom sample from a vector, dataframe or matrix on the serverside or - as a special case - randomly permutes a vector, dataframe or matrix.

Usage

```
ds.sample(
  x = NULL,
  size = NULL,
  seed.as.integer = NULL,
  replace = FALSE,
  prob = NULL,
  newobj = NULL,
  datasources = NULL,
  notify.of.progress = FALSE
)
```

Arguments

x	Either a character string providing the name for the serverside vector, matrix or data.frame to be sampled or permuted, or an integer/numeric scalar (e.g. 923) indicating that one should create a new vector on the serverside that is a randomly permuted sample of the vector 1:923, or (if [replace] = FALSE, a full random permutation of that same vector. For further details of using ds.sample with x set as an integer/numeric please see help for the `sample` function in native R. But if x is set as a character string denoting a vector, matrix or data.frame on the serverside, please note that although `ds.sample` effectively calls `sample` on the serverside it behaves somewhat differently to `sample` - for the reasons identified at the top of 'details' and so help for `sample` should be used as a guide only.
size	a numeric/integer scalar indicating the size of the sample to be drawn. If the [x] argument is a vector, matrix or data.frame on the serverside and if the [size] argument is set either to 0 or to the length of the object to be 'sampled' and [replace] is FALSE, then ds.sample will draw a random sample that includes all rows of the input object but will randomly permute them. If the [x] argument is numeric (e.g. 923) and size is either undefined or set equal to 923, the output on the serverside will be a vector of length 923 permuted into a random order. If the [replace] argument is FALSE then the value of [size] must be no greater than the length of object to be sorted - if this is violated an error message will be returned.
seed.as.integer	this is precisely equivalent to the [seed.as.integer] arguments for the pseudo-random number generating functions (e.g. also see help for ds.rBinom, ds.rNorm, ds.rPois and ds.rUnif). In other words the seed.as.integer argument is either a a numeric scalar or a NULL which primes the random seed in each data source. If <seed.as.integer> is a numeric scalar (e.g. 938) the seed in each study is set as 938*1 in the first study in the set of data sources being used, 938*2 in the second, up to 938*N in the Nth study. If <seed.as.integer> is set as 0 all sources will start with the seed value 0 and all the random number generators will therefore start from the same position. If you want to use the same starting seed in all studies but do not wish it to be 0, you can specify a non-zero scalar value for <seed.as.integer> and then use the <datasources> argument to generate the random number vectors one source at a time (e.g. ,datasources=default.opals[2] to generate the random vector in source 2). As an example, if the <seed.as.integer>

value is 78326 then the seed in each source will be set at 78326*1 = 78326 because the vector of datasources being used in each call to the function will always be of length 1 and so the source-specific seed multiplier will also be 1. The function ds.rUnif.o calls the serverside assign function setSeedDS.o to create the random seeds in each source

`replace` a Boolean indicator (TRUE or FALSE) specifying whether the sample should be drawn with or without replacement. Default is FALSE so the sample is drawn without replacement. For further details see help for `sample` in native R.

`prob` a character string containing the name of a numeric vector of probability weights on the serverside that is associated with each of the elements of the vector to be sampled enabling the drawing of a sample with some elements given higher probability of being drawn than others. For further details see help for `sample` in native R.

`newobj` This a character string providing a name for the output data.frame which defaults to 'newobj.sample' if no name is specified.

`datasources` specifies the particular opal object(s) to use. If the <datasources> argument is not specified the default set of opals will be used. The default opals are called default.opals and the default can be set using the function `ds.setDefaultOpals`. If the <datasources> is to be specified, it should be set without inverted commas: e.g. datasources=opals.em or datasources=default.opals. If you wish to apply the function solely to e.g. the second opal server in a set of three, the argument can be specified as: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set you specify: e.g. datasources=opals.em[c(1,3)]

`notify.of.progress` specifies if console output should be produce to indicate progress. The default value for notify.of.progress is FALSE.

Details

Clientside function ds.sample calls serverside assign function sampleDS. Based on the native R function `sample()` but deals slightly differently with data.frames and matrices. Specifically the `sample()` function in R identifies the length of an object and then samples n components of that length. But length(data.frame) in native R returns the number of columns not the number of rows. So if you have a data.frame with 71 rows and 10 columns, the sample() function will select 10 columns at random, which is often not what is required. So, ds.sample(x="data.frame",size=10) in DataSHIELD will sample 10 rows at random(with or without replacement depending whether the [replace] argument is TRUE or FALSE, with False being default). If x is a simple vector or a matrix it is first coerced to a data.frame on the serverside and so is dealt with in the same way (i.e. random selection of 10 rows). If x is an integer not expressed as a character string, it is dealt with in exactly the same way as in native R. That is, if x = 923 and size=117, DataSHIELD will draw a random sample in random order of size 117 from the vector 1:923 (i.e. 1, 2, ... ,923) with or without replacement depending whether [replace] is TRUE or FALSE. If the [x] argument is numeric (e.g. 923) and size is either undefined or set equal to 923, the output on the serverside will be a vector of length 923 permuted into a random order. If the [x] argument is a vector, matrix or data.frame on the serverside and if the [size] argument is set either to 0 or to the length of the object to be 'sampled' and [replace] is FALSE, then ds.sample will draw a random sample that includes all rows of the input object but will randomly permute them. This is how ds.sample enables random permuting as well as random sub-sampling. When a serverside vector, matrix or data.frame

is sampled using ds.sample 3 new columns are appended to the right of the output object. These are: 'in.sample', 'ID.seq', and 'sampling.order'. The first of these is set to 1 whenever a row enters the sample and as a QA test, all values in that column in the output object should be 1. 'ID.seq' is a sequential numeric ID appended to the right of the object to be sampled during the running of ds.sample that runs from 1 to the length of the object and will be appended even if there is already an equivalent sequential ID in the object. The output object is stored in the same original order as it was before sampling, and so if the first four elements of 'ID.seq' are 3,4, 6, 15 ... then it means that rows 1 and 2 were not included in the random sample, but rows 3, 4 were. Row 5 was not included, 6 was included and rows 7-14 were not etc. The 'sampling.order' vector is of class numeric and indicates the order in which the rows entered the sample: 1 indicates the first row sample, 2 the second etc. The lines of code that follow create an output object of the same length as the input object (PRWa) but they join the sample in random order. By sorting the output object (in this case with the default name 'newobj.sample) using ds.dataFrameSort with the 'sampling.order' vector as the sort key, the output object is rendered equivalent to PRWa but with the rows randomly permuted (so the column reflecting the vector 'sample.order' now runs from 1:length of object, while the column reflecting 'ID.seq' denoting the original order is now randomly ordered. If you need to return to the original order you can simply us ds.dataFrameSort again using the column reflecting 'ID.seq' as the sort key: (1) ds.sample('PRWa',size=0,seed.as.integer = 256); (2) ds.make("newobj.sample$sampling.order","sortkey"); (3) ds.dataFrameSort("newobj.sample","sortkey",newobj="newobj.pe
The only additional detail to note is that the original name of the sort key ("newobj.sample$sampling.order")
is 28 characters long, and because its length is tested to check for disclosure risk, this original name will fail using the usual value for 'nfilter.stringShort' (i.e. 20). This is why line 2 is inserted to create a copy with a shorter name.

Value

the object specified by the <newobj> argument (or default name 'newobj.sample') which is written to the serverside. In addition, two validity messages are returned indicating whether <newobj> has been created in each data source and if so whether it is in a valid form. If its form is not valid in at least one study - e.g. because a disclosure trap was tripped and creation of the full output object was blocked - ds.dataFrameSort() also returns any studysideMessages that may explain the error in creating the full output object. We are currently working to extend the information that can be returned to the clientside when an error occurs.

Author(s)

Paul Burton, for DataSHIELD Development Team, 15/4/2020

---

ds.scatterPlot *Generates non-disclosive scatter plots*

Description

This function uses two disclosure control methods to generate non-disclosive scatter plots of two server-side continuous variables.

Usage

```
ds.scatterPlot(
  x = NULL,
  y = NULL,
  method = "deterministic",
  k = 3,
  noise = 0.25,
  type = "split",
  return.coords = FALSE,
  datasources = NULL
)
```

Arguments

x	a character string specifying the name of the explanatory variable, a numeric vector.
y	a character string specifying the name of the response variable, a numeric vector.
method	a character string that specifies the method that is used to generated non-disclosive coordinates to be displayed in a scatter plot. This argument can be set as 'deteministic' or 'probabilistic'. Default 'deteministic'. For more information see **Details**.
k	the number of the nearest neighbours for which their centroid is calculated. Default 3. For more information see **Details**.
noise	the percentage of the initial variance that is used as the variance of the embedded noise if the argument method is set to 'probabilistic'. For more information see **Details**.
type	a character that represents the type of graph to display. This can be set as 'combine' or 'split'. Default 'split'. For more information see **Details**.
return.coords	a logical. If TRUE the coordinates of the anonymised data points are return to the Console. Default value is FALSE.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

As the generation of a scatter plot from original data is disclosive and is not permitted in DataSHIELD, this function allows the user to plot non-disclosive scatter plots.

If the argument method is set to 'deterministic', the server-side function searches for the k-1 nearest neighbours of each single data point and calculates the centroid of such k points. The proximity is defined by the minimum Euclidean distances of z-score transformed data.

When the coordinates of all centroids are estimated the function applies scaling to expand the centroids back to the dispersion of the original data. The scaling is achieved by multiplying the centroids with a scaling factor that is equal to the ratio between the standard deviation of the original variable and the standard deviation of the calculated centroids. The coordinates of the scaled centroids are then returned to the client-side.

The value of k is specified by the user. The suggested and default value is equal to 3 which is also the suggested minimum threshold that is used to prevent disclosure which is specified in the protection filter nfilter.kNN. When the value of k increases, the disclosure risk decreases but the utility loss increases. The value of k is used only if the argument method is set to 'deterministic'. Any value of k is ignored if the argument method is set to 'probabilistic'.

If the argument method is set to 'probabilistic', the server-side function generates a random normal noise of zero mean and variance equal to 10% of the variance of each x and y variable. The noise is added to each x and y variable and the disturbed by the addition of noise data are returned to the client-side. Note that the seed random number generator is fixed to a specific number generated from the data and therefore the user gets the same figure every time that chooses the probabilistic method in a given set of variables. The value of noise is used only if the argument method is set to 'probabilistic'. Any value of noise is ignored if the argument method is set to 'deterministic'.

In type argument can be set two graphics to display:
(1) If type = 'combine' a scatter plot for combined data is generated.
(2) If type = 'split' one scatter plot for each study is generated.

Server function called: scatterPlotDS

Value

ds.scatterPlot returns to the client-side one or more scatter plots depending on the argument type.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
```

```
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()
  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Example 1: generate a scatter plot for each study separately
  #Using the default deterministic method and k = 10

  ds.scatterPlot(x = "D$PM_BMI_CONTINUOUS",
                y = "D$LAB_GLUC_ADJUSTED",
                method = "deterministic",
                k = 10,
                type = "split",
                datasources = connections)

  #Example 2: generate a combined scatter plot with the probabilistic method
  #and noise of variance 0.5% of the variable's variance, and display the coordinates
  # of the anonymised data points to the Console

  ds.scatterPlot(x = "D$PM_BMI_CONTINUOUS",
                y = "D$LAB_GLUC_ADJUSTED",
                method = "probabilistic",
                noise = 0.5,
                type = "combine",
                datasources = connections)

  #Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.seq *Generates a sequence in the server-side*

Description

This function generates a sequence for given parameters on the server-side.

Usage

```
ds.seq(
  FROM.value.char = "1",
  BY.value.char = "1",
  TO.value.char = NULL,
  LENGTH.OUT.value.char = NULL,
  ALONG.WITH.name = NULL,
  newobj = "newObj",
  datasources = NULL
)
```

Arguments

FROM.value.char	an integer or a number in character from specifying the starting value for the sequence. Default "1".
BY.value.char	an integer or a number in character from specifying the value to increment each step in the sequence. Default "1".
TO.value.char	an integer or a number in character from specifying the terminal value for the sequence. Default NULL. For more information see **Details**.
LENGTH.OUT.value.char	an integer or a number in character from specifying the length of the sequence at which point its extension should be stopped. Default NULL. For more information see **Details**.
ALONG.WITH.name	a character string specifying the name of a standard vector to generate a vector of the same length. For more information see **Details**.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default seq.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to a native R function seq(). It creates a flexible range of sequence vectors that can then be used to help manage and analyse data.

Note: the combinations of arguments that are not allowed for the function seq in native R are also prohibited in ds.seq.

To be specific, FROM.value.char argument defines the start of the sequence and BY.value.char defines how the sequence is incremented (or decremented) at each step. But where the sequence stops can be defined in three different ways:
(1) TO.value.char indicates the terminal value of the sequence. For example, ds.seq(FROM.value.char = "3", BY.value.char = "2",TO.value.char = "7") creates the sequence 3,5,7 on the server-side.
(2) LENGTH.OUT.value.char indicates the length of the sequence. For example, ds.seq(FROM.value.char = "3", BY.value.char = "2",LENGTH.OUT.value.char = "7") creates the sequence 3,5,7,9,11,13,15 on the server-side.
(3) ALONG.WITH.name specifies the name of a variable on the server-side, such that the sequence in each study will be equal in length to that variable. For example, ds.seq(FROM.value.char = "3", BY.value.char = "2",ALONG.WITH.name = "var.x") creates a sequence such that if var.x is of length 100 in study 1 the sequence written to study 1 will be 3,5,7,...,197,199,201 and if var.x is of length 4 in study 2, the sequence written to study 2 will be 3,5,7,9.
Only one of the three arguments: TO.value.char, LENGTH.OUT.value.char and ALONG.WITH.name can be non-null in any one call.

In LENGTH.OUT.value.char argument if you specify a number with a decimal point but in character form this result in a sequence length(integer) + 1. For example, LENGTH.OUT.value.char = "1000.0001" generates a sequence of length 1001.

Server function called: seqDS

Value

ds.seq returns to the server-side the generated sequence. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki
  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Create 3 different sequences

  ds.seq(FROM.value.char = "1",
         BY.value.char = "2",
         TO.value.char = "7",
         newobj = "new.seq1",
         datasources = connections)


  ds.seq(FROM.value.char = "4",
         BY.value.char = "3",
         LENGTH.OUT.value.char = "10",
         newobj = "new.seq2",
         datasources = connections)
```

```
  ds.seq(FROM.value.char = "2",
         BY.value.char = "5",
         ALONG.WITH.name = "D$GENDER",
         newobj = "new.seq3",
         datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.setSeed` *Server-side random number generation*

Description

Primes the pseudorandom number generator in a data source

Usage

```
ds.setSeed(seed.as.integer = NULL, datasources = NULL)
```

Arguments

`seed.as.integer`	a numeric value or a NULL that primes the random seed in each data source.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function generates an instance of the full pseudorandom number seed that is a vector of integers of length 626 called `.Random.seed`, this vector is written to the server-side.

This function is similar to a native R function `set.seed()`.

In `seed.as.integer` argument the current limitation on the value of the integer that can be specified is `-2147483647` up to `+2147483647` (this is `+/- ([2^31]-1)`).

Because you only specify one integer in the call to `ds.setSeed` (i.e. the value for the `seed.as.integer` argument) that value will be used as the priming trigger value in all of the specified data sources and so the pseudorandom number generators will all start from the same position and if a vector of pseudorandom number values is requested based on one of DataSHIELD's pseudorandom number generating functions precisely the same random vector will be generated in each source. If you want to avoid this you can specify a different priming value in each source by using the `datasources` argument to generate the random number vectors one source at a time with a different integer in each case.

Furthermore, if you use any one of DataSHIELD's pseudorandom number generating functions: `ds.rNorm`, `ds.rUnif`, `ds.rPois` or `ds.rBinom`. The function call itself automatically uses the single integer priming seed you specify to generate different integers in each source.

Server function called: `setSeedDS`

Value

Sets the values of the vector of integers of length 626 known as `.Random.seed` on each data source that is the true current state of the random seed in each source. It also returns the value of the trigger integer that has primed the random seed vector (`.Random.seed`) in each source and also the integer vector of 626 elements that is `.Random.seed itself`.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Generate a pseudorandom number in the server-side

  ds.setSeed(seed.as.integer = 152584,
             datasources = connections)

  #Specify the pseudorandom number only in the first source
```

```
  ds.setSeed(seed.as.integer = 741,
             datasources = connections[1])#only the frist study is used (study1)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.skewness *Calculates the skewness of a server-side numeric variable*

Description

This function calculates the skewness of a numeric variable that is stored on the server-side (Opal server).

Usage

```
ds.skewness(x = NULL, method = 1, type = "both", datasources = NULL)
```

Arguments

x	a character string specifying the name of a numeric variable.
method	an integer value between 1 and 3 selecting one of the algorithms for computing skewness. For more information see **Details**. The default value is set to 1.
type	a character string which represents the type of analysis to carry out. type can be set as: 'combine', 'split' or 'both'. For more information see **Details**. The default value is set to 'both'.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the function skewness in R package e1071.

The function calculates the skewness of an input variable x with three different methods:
(1) If method is set to 1 the following formula is used $skewness = \frac{\sum_{i=1}^{N}(x_i-\bar{(}x))^3/N}{(\sum_{i=1}^{N}((x_i-\bar{(}x))^2)/N)^(3/2)}$, where $\bar{x}$ is the mean of x and N is the number of observations.
(2) If method is set to 2 the following formula is used $skewness = \frac{\sum_{i=1}^{N}(x_i-\bar{(}x))^3/N}{(\sum_{i=1}^{N}((x_i-\bar{(}x))^2)/N)^(3/2)} * \frac{\sqrt{(}N(N-1)}{n-2}$.
(3) If method is set to 3 the following formula is used $skewness = \frac{\sum_{i=1}^{N}(x_i-\bar{(}x))^3/N}{(\sum_{i=1}^{N}((x_i-\bar{(}x))^2)/N)^(3/2)} * (\frac{N-1}{N})^(3/2)$.

The type argument can be set as follows:
(1) If type is set to 'combine', 'combined', 'combines' or 'c', the global skewness is returned.

(2) If type is set to 'split', 'splits' or 's', the skewness is returned separately for each study.
(3) If type is set to 'both' or 'b', both sets of outputs are produced.

If x contains any missing value, the function removes those before the calculation of the skewness.

Server functions called: skewnessDS1 and skewnessDS2

Value

ds.skewness returns a matrix showing the skewness of the input numeric variable, the number of valid observations and the validity message.

Author(s)

Demetris Avraam, for DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

 #Calculate the skewness of LAB_TSC numeric variable for each study separately and combined

  ds.skewness(x = "D$LAB_TSC",
              method = 1,
              type = "both",
             datasources = connections)

  # Clear the Datashield R sessions and logout
```

```
  DSI::datashield.logout(connections)

## End(Not run)
```

`ds.sqrt` *Computes the square root values of a variable*

Description

Computes the square root values for a specified numeric or integer vector. This function is similar to R function `sqrt`.

Usage

```
ds.sqrt(x = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x`	a character string providing the name of a numeric or an integer vector.
`newobj`	a character string that provides the name for the output variable that is stored on the data servers. Default name is set to `sqrt.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

The function calls the server-side function `sqrtDS` that computes the square root values of the elements of a numeric or integer vector and assigns a new vector with those square root values on the server-side. The name of the new generated vector is specified by the user through the argument `newobj`, otherwise is named by default to `sqrt.newobj`.

Value

`ds.sqrt` assigns a vector for each study that includes the square root values of the input numeric or integer vector specified in the argument `x`. The created vectors are stored in the servers.

Author(s)

Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")

  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Example 1: Get the square root of LAB_HDL variable
  ds.sqrt(x='D$LAB_HDL', newobj='LAB_HDL.sqrt', datasources=connections)
  # compare the mean of LAB_HDL and of LAB_HDL.sqrt
  # Note here that the number of missing values is bigger in the LAB_HDL.sqrt
  ds.mean(x='D$LAB_HDL', datasources=connections)
  ds.mean(x='LAB_HDL.sqrt', datasources=connections)

  # Example 2: Generate a repeated vector of the squares of integers from 1 to 10
  # and get their square roots
 ds.make(toAssign='rep((1:10)^2, times=10)', newobj='squares.vector', datasources=connections)
  ds.sqrt(x='squares.vector', newobj='sqrt.vector', datasources=connections)
  ds.table(rvar='squares.vector')$output.list$TABLE_rvar.by.study_counts
  ds.table(rvar='sqrt.vector')$output.list$TABLE_rvar.by.study_counts

  # clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.subset *Generates a valid subset of a table or a vector*

Description

The function uses the R classical subsetting with squared brackets '[]' and allows also to subset using a logical operator and a threshold. The object to subset from must be a vector (factor, numeric or character) or a table (data.frame or matrix).

Usage

```
ds.subset(
  x = NULL,
  subset = "subsetObject",
  completeCases = FALSE,
  rows = NULL,
  cols = NULL,
  logicalOperator = NULL,
  threshold = NULL,
  datasources = NULL
)
```

Arguments

x	a character, the name of the dataframe or the factor vector and the range of the subset.
subset	the name of the output object, a list that holds the subset object. If set to NULL the default name of this list is 'subsetObject'
completeCases	a character that tells if only complete cases should be included or not.
rows	a vector of integers, the indices of the rows to extract.
cols	a vector of integers or a vector of characters; the indices of the columns to extract or their names.
logicalOperator	a boolean, the logical parameter to use if the user wishes to subset a vector using a logical operator. This parameter is ignored if the input data is not a vector.
threshold	a numeric, the threshold to use in conjunction with the logical parameter. This parameter is ignored if the input data is not a vector.
datasources	a list of DSConnection-class objects obtained after login. If the <datasources> the default set of connections will be used: see datashield.connections_default.

Details

(1) If the input data is a table the user specifies the rows and/or columns to include in the subset; the columns can be referred to by their names. Table subsetting can also be done using the name of a variable and a threshold (see example 3). (2) If the input data is a vector and the parameters 'rows', 'logical' and 'threshold' are all provided the last two are ignored (i.e. 'rows' has precedence over the other two parameters then). IMPORTANT NOTE: If the requested subset is not valid (i.e. contains less than the allowed number of observations) all the values are turned into missing values (NA). Hence an invalid subset is indicated by the fact that all values within it are set to NA.

Value

no data are return to the user, the generated subset dataframe is stored on the server side.

Author(s)

Gaye, A.

See Also

ds.subsetByClass to subset by the classes of factor vector(s).

ds.meanByClass to compute mean and standard deviation across categories of a factor vectors.

Examples

```
## Not run:

  # load the login data
  data(logindata)

  # login and assign some variables to R
  myvar <- list("DIS_DIAB","PM_BMI_CONTINUOUS","LAB_HDL", "GENDER")
  conns <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

 # Example 1: generate a subset of the assigned dataframe (by default the table is named 'D')
  # with complete cases only
  ds.subset(x='D', subset='subD1', completeCases=TRUE)
 # display the dimensions of the initial table ('D') and those of the subset table ('subD1')
  ds.dim('D')
  ds.dim('subD1')

  # Example 2: generate a subset of the assigned table (by default the table is named 'D')
  # with only the variables
  # DIS_DIAB' and'PM_BMI_CONTINUOUS' specified by their name.
  ds.subset(x='D', subset='subD2', cols=c('DIS_DIAB','PM_BMI_CONTINUOUS'))

 # Example 3: generate a subset of the table D with bmi values greater than or equal to 25.
  ds.subset(x='D', subset='subD3', logicalOperator='PM_BMI_CONTINUOUS>=', threshold=25)

  # Example 4: get the variable 'PM_BMI_CONTINUOUS' from the dataframe 'D' and generate a
  # subset bmi
  # vector with bmi values greater than or equal to 25
  ds.assign(toAssign='D$PM_BMI_CONTINUOUS', newobj='BMI')
  ds.subset(x='BMI', subset='BMI25plus', logicalOperator='>=', threshold=25)

  # Example 5: subsetting by rows:
  # get the logarithmic values of the variable 'lab_hdl' and generate a subset with
  # the first 50 observations of that new vector. If the specified number of row is
  # greater than the total
  # number of rows in any of the studies the process will stop.
  ds.assign(toAssign='log(D$LAB_HDL)', newobj='logHDL')
  ds.subset(x='logHDL', subset='subLAB_HDL', rows=c(1:50))
```

```
  # now get a subset of the table 'D' with just the 100 first observations
  ds.subset(x='D', subset='subD5', rows=c(1:100))

  # clear the Datashield R sessions and logout
  datashield.logout(conns)


## End(Not run)
```

`ds.subsetByClass` *Generates valid subset(s) of a data frame or a factor*

Description

The function takes a categorical variable or a data frame as input and generates subset(s) variables or data frames for each category.

Usage

```
ds.subsetByClass(
  x = NULL,
  subsets = "subClasses",
  variables = NULL,
  datasources = NULL
)
```

Arguments

`x`	a character, the name of the dataframe or the vector to generate subsets from.
`subsets`	the name of the output object, a list that holds the subset objects. If set to NULL the default name of this list is 'subClasses'.
`variables`	a vector of string characters, the name(s) of the variables to subset by.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the <datasources> the default set of connections will be used: see datashield.connections_default.

Details

If the input data object is a data frame it is possible to specify the variables to subset on. If a subset is not 'valid' all its the values are reported as missing (i.e. NA), the name of the subsets is labelled with the suffix '_INVALID'. Subsets are considered invalid if the number of observations it holds are between 1 and the threshold allowed by the data owner. if a subset is empty (i.e. no entries) the name of the subset is labelled with the suffix '_EMPTY'.

Value

a no data are return to the user but messages are printed out.

Author(s)

Gaye, A.

See Also

ds.meanByClass to compute mean and standard deviation across categories of a factor vectors.

ds.subset to subset by complete cases (i.e. removing missing values), threshold, columns and rows.

Examples

```
## Not run:

  # load the login data
  data(logindata)

  # login and assign some variables to R
  myvar <- list('DIS_DIAB','PM_BMI_CONTINUOUS','LAB_HDL', 'GENDER')
  conns <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

 # Example 1: generate all possible subsets from the table assigned above (one subset table
  # for each class in each factor)
  ds.subsetByClass(x='D', subsets='subclasses')
  # display the names of the subset tables that were generated in each study
  ds.names('subclasses')

  # Example 2: subset the table initially assigned by the variable 'GENDER'
  ds.subsetByClass(x='D', subsets='subtables', variables='GENDER')
  # display the names of the subset tables that were generated in each study
  ds.names('subtables')

  # Example 3: generate a new variable 'gender' and split it into two vectors: males
  # and females
  ds.assign(toAssign='D$GENDER', newobj='gender')
  ds.subsetByClass(x='gender', subsets='subvectors')
  # display the names of the subset vectors that were generated in each study
  ds.names('subvectors')

  # clear the Datashield R sessions and logout
  datashield.logout(conns)


## End(Not run)
```

`ds.summary` *Generates the summary of a server-side object*

Description

Generates the summary of a server-side object.

Usage

```
ds.summary(x = NULL, datasources = NULL)
```

Arguments

x	a character string specifying the name of a numeric or factor variable.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function provides some insight about an object. Unlike the similar native R summary function only a limited class of objects can be used as input to reduce the risk of disclosure. For example, the minimum and the maximum values of a numeric vector are not given to the client because they are potentially disclosive.

server functions called: isValidDS, dimDS and colnamesDS

Value

ds.summary returns to the client-side the class and size of the server-side object. Also other information is returned depending on the class of the object. For example, potentially disclosive information such as the minimum and maximum values of numeric vectors are not returned. The summary is given for each study separately.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

  ## Version 6, for version 5 see the Wiki

  # Connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
```

```
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  # Log onto the remote Opal training servers
  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Calculate the summary of a numeric variable

  ds.summary(x = "D$LAB_TSC",
             datasources = connections)

  #Calculate the summary of a factor variable

  ds.summary(x = "D$PM_BMI_CATEGORICAL",
             datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.table` *Generates 1-, 2-, and 3-dimensional contingency tables with option of assigning to serverside only and producing chi-squared statistics*

Description

Creates 1-dimensional, 2-dimensional and 3-dimensional tables using the `table` function in native R.

Usage

```
ds.table(
  rvar = NULL,
  cvar = NULL,
  stvar = NULL,
  report.chisq.tests = FALSE,
  exclude = NULL,
  useNA = "always",
  suppress.chisq.warnings = FALSE,
  table.assign = FALSE,
  newobj = NULL,
  datasources = NULL,
  force.nfilter = NULL
)
```

Arguments

rvar	is a character string (in inverted commas) specifying the name of the variable defining the rows in all of the 2 dimensional tables that form the output. Please see 'details' above for more information about one-dimensional tables when a variable name is provided by <rvar> but <cvar> and <stvar> are both NULL
cvar	is a character string specifying the name of the variable defining the columns in all of the 2 dimensional tables that form the output.
stvar	is a character string specifying the name of the variable that indexes the separate two dimensional tables in the output if the call specifies a 3 dimensional table.
report.chisq.tests	if TRUE, chi-squared tests are applied to every 2 dimensional table in the output and reported as "chisq.test_table.name". Default = FALSE.
exclude	this argument is passed through to the table function in native R which is called by tableDS. The help for table in native R indicates that 'exclude' specifies any levels that should be deleted for all factors in rvar, cvar or stvar. If the <exclude> argument does not include NA and if the <useNA> argument is not specified, it implies <useNA> = "always" in DataSHIELD. If you read the help for table in native R including the 'details' and the 'examples' (particularly 'd.patho') you will see that the response of table to different combinations of the <exclude> and <useNA> arguments can be non-intuitive. This is particularly so if there is more than one type of missing (e.g. missing by observation as well as missing because of an NaN response to a mathematical function - such as log(-3.0)). In DataSHIELD, if you are in one of these complex settings (which should not be very common) and you cannot interpret the output that has been approached you might try: (1) making sure that the variable producing the strange results is of class factor rather than integer or numeric - although integers and numerics are coerced to factors by ds.table they can occasionally behave less well when the NA setting is complex; (2) specify both an <exclude> argument e.g. exclude = c("NaN","3") and a <useNA> argument e.g. useNA= "no"; (3) if you are excluding multiple levels e.g exclude = c("NA","3") then you can reduce this to one e.g. exclude = c("NA") and then remove the 3s by deleting rows of data, or converting the 3s to a different value.
useNA	this argument is passed through to the table function in native R which is called by tableDS. In DataSHIELD, this argument can take two values: "no" or "always" which indicate whether to include NA values in the table. For further information, please see the help for the <exclude> argument (above) and/or the help for the table function in native R. Default value is set to "always".
suppress.chisq.warnings	if set to TRUE, the default warnings are suppressed that would otherwise be produced by the table function in native R whenever an expected cell count in one or more cells is less than 5. Default is FALSE. Further details can be found under 'details' and the help provided for the <report.chisq.tests> argument (above).
table.assign	is a Boolean argument set by default to FALSE. If it is FALSE the ds.table function acts as a standard aggregate function - it returns the table that is specified in its call to the clientside where it can be visualised and worked with by

the analyst. But if <table.assign> is TRUE, the same table object is also written to the serverside. As explained under 'details' (above), this may be useful when some elements of a table need to be used to drive forward the overall analysis (e.g. to help select individuals for an analysis sub-sample), but the required table cannot be visualised or returned to the clientside because it fails disclosure rules.

`newobj` this a character string providing a name for the output table object to be written to the serverside if <table.assign> is TRUE. If no explicit name for the table object is specified, but <table.assign> is nevertheless TRUE, the name for the serverside table object defaults to `table.newobj`.

`datasources` a list of `DSConnection-class` objects obtained after login. If the <datasources> the default set of connections will be used: see datashield.connections_default. If the <datasources> is to be specified, it should be set without inverted commas: e.g. datasources=connections.em or datasources=default.connections. If you wish to apply the function solely to e.g. the second connection server in a set of three, the argument can be specified as: e.g. datasources=connections.em[2]. If you wish to specify the first and third connection servers in a set you specify: e.g. datasources=connections.em[c(1,3)].

`force.nfilter` if <force.nfilter> is non-NULL it must be specified as a positive integer represented as a character string: e.g. "173". This the has the effect of the standard value of 'nfilter.tab' (often 1, 3, 5 or 10 depending what value the data custodian has selected for this particular data set), to this new value (here, 173). CRUCIALLY, the `ds.table` function only allows the standard value to be INCREASED. So if the standard value has been set as 5 (as one of the R options set in the serverside connection), "6" and "4981" would be allowable values for the <force.nfilter> argument but "4" or "1" would not. The purpose of this argument is for the user or developer to force the table to fail the disclosure control tests so the he/she can see what then happens and check that it is behaving as anticipated/hoped.

Details

The `ds.table` function selects numeric, integer or factor variables on the serverside which define a contingency table with up to three dimensions. The native R `table` function basically operates on factors and if variables are specified that are integers or numerics they are first coerced to factors. If the 1-dimensional, 2-dimensional or 3-dimensional table generated from a given study satisfies appropriate disclosure-control criteria it can be returned directly to the clientside where it is presented as a study-specific table and is also included in a combined table across all studies.

The data custodian responsible for data security in a given study can specify the minimum non-zero cell count that determines whether the disclosure-control criterion can be viewed as having been met. If the count in any one cell in a table falls below the specified threshold (and is also non-zero) the whole table is blocked and cannot be returned to the clientside. However, even if a table is potentially disclosive it can still be written to the serverside while an empty representation of the structure of the table is returned to the clientside. The contents of the cells in the serverside table object are reflected in a vector of counts which is one component of that table object.

The true counts in the studyside vector are replaced by a sequential set of cell-IDs running from 1:n (where n is the total number of cells in the table) in the empty representation of the structure

of the potentially disclosive table that is returned to the clientside. These cell-IDs reflect the order of the counts in the true counts vector on the serverside. In consequence, if the number 13 appears in a cell of the empty table returned to the clientside, it means that the true count in that same cell is held as the 13th element of the true count vector saved on the serverside. This means that a data analyst can still make use of the counts from a call to the `ds.table` function to drive their ongoing analysis even when one or more non-zero cell counts fall below the specified threshold for potential disclosure risk.

Because the table object on the serverside cannot be visualised or transferred to the clientside, DataSHIELD ensures that although it can, in this way, be used to advance analysis, it does not create a direct risk of disclosure.

The <rvar> argument identifies the variable defining the rows in each of the 2-dimensional tables produced in the output.

The <cvar> argument identifies the variable defining the columns in the 2-dimensional tables produced in the output.

In creating a 3-dimensional table the <stvar> ('separate tables') argument identifies the variable that indexes the set of two dimensional tables in the output `ds.table`.

As a minor technicality, it should be noted that if a 1-dimensional table is required, one only need specify a value for the <rvar> argument and any one dimensional table in the output is presented as a row vectors and so technically the <rvar> variable defines the columns in that 1 x n vector. However, the ds.table function deals with 1-dimensional tables differently to 2 and 3 dimensional tables and key components of the output for one dimensional tables are actually two dimensional: with rows defined by <rvar> and with one column for each of the studies.

The output list generated by `ds.table` contains tables based on counts named "table.name_counts" and other tables reporting corresponding column proportions ("table.name_col.props") or row proportions ("table.name_row.props"). In one dimensional tables in the output the output tables include _counts and _proportions. The latter are not called _col.props or _row.props because, for the reasons noted above, they are technically column proportions but are based on the distribution of the <rvar> variable.

If the <report.chisq.tests> argument is set to TRUE, chisq tests are applied to every 2-dimensional table in the output and reported as "chisq.test_table.name". The <report.chisq.tests> argument defaults to FALSE.

If there is at least one expected cell counts < 5 in an output table, the native R <chisq.test> function returns a warning. Because in a DataSHIELD setting this often means that every study and several tables may return the same warning and because it is debatable whether this warning is really statistically important, the <suppress.chisq.warnings> argument can be set to TRUE to block the warnings. However, it is defaulted to FALSE.

Value

Having created the requested table based on serverside data it is returned to the clientside for the analyst to visualise (unless it is blocked because it fails the disclosure control criteria or there is an error for some other reason).

The clientside output from `ds.table` includes error messages that identify when the creation of a table from a particular study has failed and why. If table.assign=TRUE, `ds.table` also writes the requested table as an object named by the <newobj> argument or set to 'newObj' by default.

Further information about the visible material passed to the clientside, and the optional table object written to the serverside can be seen under 'details' (above).

Author(s)

Paul Burton and Alex Westerberg for DataSHIELD Development Team, 01/05/2020

`ds.table1D` *Generates 1-dimensional contingency tables*

Description

The function ds.table1D is a client-side wrapper function. It calls the server-side function table1DDS to generate 1-dimensional tables for all data sources.

Usage

```
ds.table1D(
  x = NULL,
  type = "combine",
  warningMessage = TRUE,
  datasources = NULL
)
```

Arguments

`x`	a character, the name of a numerical vector with discrete values - usually a factor.
`type`	a character which represent the type of table to output: pooled table or one table for each data source. If `type` is set to 'combine', a pooled 1-dimensional table is returned; if If `type` is set to 'split' a 1-dimensional table is returned for each data source.
`warningMessage`	a boolean, if set to TRUE (default) a warning is displayed if any returned table is invalid. Warning messages are suppressed if this parameter is set to FALSE. However the analyst can still view 'validity' information which are stored in the output object 'validity' - see the list of output objects.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the <datasources> the default set of connections will be used: see datashield.connections_default.

Details

The table returned by the server side function might be valid (non disclosive - no table cell have counts between 1 and the minimal number agreed by the data owner and set in the data repository) or invalid (potentially disclosive - one or more table cells have a count between 1 and the minimal number agreed by the data owner). If a 1-dimensional table is invalid all the cells are set to NA except the total count. This way it is possible the know the total count and combine total counts across data sources but it is not possible to identify the cell(s) that had the small counts which render the table invalid.

Value

A list object containing the following items:

counts	table(s) that hold counts for each level/category. If some cells counts are invalid (see 'Details' section) only the total (outer) cell counts are displayed in the returned individual study tables or in the pooled table.
percentages	table(s) that hold percentages for each level/category. Here also inner cells are reported as missing if one or more cells are 'invalid'.
validity	a text that informs the analyst about the validity of the output tables. If any tables are invalid the studies they are originated from are also mentioned in the text message.

Author(s)

Gaye, A.; Burton, P.

See Also

ds.table2D for cross-tabulating two vectors.

Examples

```
## Not run:

  # load the file that contains the login details
  data(logindata)

  # login and assign all the stored variables to R
  conns <- datashield.login(logins=logindata,assign=TRUE)

 # Example 1: generate a one dimensional table, outputting combined (pooled) contingency tables
  output <- ds.table1D(x='D$GENDER')
  output$counts
  output$percentages
  output$validity

 # Example 2: generate a one dimensional table, outputting study specific contingency tables
  output <- ds.table1D(x='D$GENDER', type='split')
  output$counts
  output$percentages
  output$validity

  # Example 3: generate a one dimensional table, outputting study specific and combined
  # contingency tables - see what happens if the reruened table is 'invalid'.
  output <- ds.table1D(x='D$DIS_CVA')
  output$counts
  output$percentages
  output$validity

  # clear the Datashield R sessions and logout
  datashield.logout(conns)
```

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

`ds.table2D` *Generates 2-dimensional contingency tables*

Description

The function ds.table2D is a client-side wrapper function. It calls the server-side function 'table2DDS' that generates a 2-dimensional contingency table for each data source.

Usage

```
ds.table2D(
  x = NULL,
  y = NULL,
  type = "both",
  warningMessage = TRUE,
  datasources = NULL
)
```

Arguments

`x`	a character, the name of a numerical vector with discrete values - usually a factor.
`y`	a character, the name of a numerical vector with discrete values - usually a factor.
`type`	a character which represent the type of table to output: pooled table or one table for each data source or both. If `type` is set to 'combine', a pooled 2-dimensional table is returned; If `type` is set to 'split' a 2-dimensional table is returned for each data source. If `type` is set to 'both' (default) a pooled 2-dimensional table plus a 2-dimensional table for each data source are returned.
`warningMessage`	a boolean, if set to TRUE (default) a warning is displayed if any returned table is invalid. Warning messages are suppressed if this parameter is set to FALSE. However the analyst can still view 'validity' information which are stored in the output object 'validity' - see the list of output objects.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the <datasources> the default set of connections will be used: see datashield.connections_default.

Details

The table returned by the server side function might be valid (non disclosive - no table cell have counts between 1 and the minimal number agreed by the data owner and set in the data repository as the "nfilter.tab") or invalid (potentially disclosive - one or more table cells have a count between 1 and the minimal number agreed by the data owner). If a 2-dimensional table is invalid all the cells are set to NA except the total counts. In this way, it is possible to combine total counts across all the data sources but it is not possible to identify the cell(s) that had the small counts which render the table invalid.

Value

A list object containing the following items:

colPercent	table(s) that hold column percentages for each level/category. Inner cells are reported as missing if one or more cells are 'invalid'.
rowPercent	table(s) that hold row percentages for each level/category. Inner cells are reported as missing if one or more cells are 'invalid'.
chi2Test	Chi-squared test for homogeneity.
counts	table(s) that hold counts for each level/category. If some cell counts are invalid (see 'Details' section) only the total (outer) cell counts are displayed in the returned individual study tables or in the pooled table.
validity	a text that informs the analyst about the validity of the output tables. If any tables are invalid the studies they are originated from are also mentioned in the text message.

Author(s)

Amadou Gaye, Paul Burton, Demetris Avraam, for DataSHIELD Development Team

See Also

ds.table1D for the tabulating one vector.

Examples

```
## Not run:

  # load the file that contains the login details
  data(logindata)

  # login and assign all the variables to R
  conns <- datashield.login(logins=logindata,assign=TRUE)

  # Example 1: generate a two dimensional table, outputting combined contingency
  # tables - default behaviour
  output <- ds.table2D(x='D$DIS_DIAB', y='D$GENDER')
  # display the 5 results items, one at a time to avoid having too much information
  # displayed at the same time
  output$counts
  output$rowPercent
  output$colPercent
  output$chi2Test
  output$validity

 # Example 2: generate a two dimensional table, outputting study specific contingency tables
  ds.table2D(x='D$DIS_DIAB', y='D$GENDER', type='split')
 # display the 5 results items, one at a time to avoid having too much information displayed
  # at the same time
  output$counts
  output$rowPercent
```

```
  output$colPercent
  output$chi2Test
  output$validity

  # Example 3: generate a two dimensional table, outputting combined contingency tables
  # *** this example shows what happens when one or studies return an invalid table ***
  output <- ds.table2D(x='D$DIS_CVA', y='D$GENDER', type='combine')
  output$counts
  output$rowPercent
  output$colPercent
  output$chi2Test
  output$validity

  # Example 4: same example as above but output is given for each study,
  # separately (i.e. type='split')
  # *** this example shows what happens when one or studies return an invalid table ***
  output <- ds.table2D(x='D$DIS_CVA', y='D$GENDER', type='split')
  output$counts
  output$rowPercent
  output$colPercent
  output$chi2Test
  output$validity

  # clear the Datashield R sessions and logout
  datashield.logout(conns)


## End(Not run)
```

---

ds.tapply *Applies a Function Over a Server-Side Ragged Array*

Description

Apply one of a selected range of functions to summarize an outcome variable over one or more indexing factors. The resultant summary is written to the client-side.

Usage

```
ds.tapply(
  X.name = NULL,
  INDEX.names = NULL,
  FUN.name = NULL,
  datasources = NULL
)
```

Arguments

X.name	a character string specifying the name of the variable to be summarized.
INDEX.names	a character string specifying the name of a single factor or a list or vector of names of up to two factors to index the variable to be summarized. For more information see **Details**.
FUN.name	a character string specifying the name of one of the allowable summarizing functions. This can be set as: "N" (or "length"), "mean","sd", "sum", or "quantile". For more information see **Details**.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to a native R function tapply(). It applies one of a selected range of functions to each cell of a ragged array, that is to each (non-empty) group of values given by each unique combination of a series of indexing factors.

The range of allowable summarizing functions for DataSHIELD ds.tapply function is much more restrictive than for the native R tapply function. The reason for this is the protection against disclosure risk.

Should other functions be required in the future then, provided they are non-disclosive, the DataSHIELD development team could work on them if requested.

To protect against disclosure the number of observations in each summarizing group in each source is calculated and if any of these falls below the value of nfilter.tab (the minimum allowable non-zero count in a contingency table) the tapply analysis of that source will return only an error message. The value of nfilter.tab is can be set and modified only by the data custodian. If an analytic team wishes the value to be reduced (e.g. to 1 which will allow any output from tapply to be returned) this needs to formally be discussed and agreed with the data custodian.

If the reason for the tapply analysis is, for example, to break a dataset down into a small number of values for each individual and then to flag up which individuals have got at least one positive value for a binary outcome variable, then that flagging does not have to be overtly returned to the client-side. Rather, it can be written as a vector to the server-side at each source (which, like any other server-side object, cannot then be seen, abstracted or copied). This can be done using ds.tapply.assign which writes the results as a newobj to the server-side and does not test the number of observations in each group against nfilter.tab. For more information see the help option of ds.tapply.assign function.

The native R tapply function has optional arguments such as na.rm = TRUE for FUN = mean which will exclude any NAs from the outcome variable to be summarized. However, in order to keep DataSHIELD's ds.tapply and ds.tapply.assign functions straightforward, the server-side functions tapplyDS and tapplyDS.assign both starts by stripping any observations which have missing (NA) values in either the outcome variable or in any one of the indexing factors. In consequence, the resultant analyses are always based on complete cases.

In INDEX.names argument the native R tapply function can coerce non-factor vectors into factors. However, this does not always work when using the DataSHIELD ds.tapply or ds.tapply.assign functions so if you are concerned that an indexing vector is not being treated correctly as a factor, please first declare it explicitly as a factor using ds.asFactor.

In FUN.name argument the allowable functions are: N or length (the number of (non-missing) observations in the group defined by each combination of indexing factors); mean; SD (standard deviation); sum; quantile (with quantile probabilities set at c(0.05,0.1,0.2,0.25,0.3,0.33,0.4,0.5,0.6,0.67,0.7,0.75,0.8,0.9,0.95)).

Server function called: tapplyDS

Value

ds.tapply returns to the client-side an array of the summarized values. It has the same number of dimensions as INDEX.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Apply a Function Over a Server-Side Ragged Array

  ds.assign(toAssign = "D$LAB_TSC",
            newobj = "LAB_TSC",
            datasources = connections)

  ds.assign(toAssign = "D$GENDER",
            newobj =  "GENDER",
            datasources = connections)
```

```
  ds.tapply(X.name = "LAB_TSC",
            INDEX.names = c("GENDER"),
            FUN.name = "mean",
            datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

`ds.tapply.assign` *Applies a Function Over a Ragged Array on the server-side*

Description

Applies one of a selected range of functions to summarize an outcome variable over one or more indexing factors and write the resultant summary as an object on the server-side.

Usage

```
ds.tapply.assign(
  X.name = NULL,
  INDEX.names = NULL,
  FUN.name = NULL,
  newobj = NULL,
  datasources = NULL
)
```

Arguments

`X.name`	a character string specifying the name of the variable to be summarized.
`INDEX.names`	a character string specifying the name of a single factor or a vector of names of up to two factors to index the variable to be summarized. For more information see **Details**.
`FUN.name`	a character string specifying the name of one of the allowable summarizing functions. This can be set as: `"N"` (or `"length"`), `"mean"`,`"sd"`, `"sum"`, or `"quantile"`. For more information see **Details**.
`newobj`	a character string that provides the name for the output variable that is stored on the data servers. Default `tapply.assign.newobj`.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the `datasources` argument is not specified the default set of connections will be used: see `datashield.connections_defa`

Details

This function applies one of a selected range of functions to each cell of a ragged array, that is to each (non-empty) group of values given by each unique combination of a series of indexing factors.

The range of allowable summarizing functions for DataSHIELD ds.tapply function is much more restrictive than for the native R tapply function. The reason for this is the protection against disclosure risk.

Should other functions be required in the future then, provided they are non-disclosive, the DataSHIELD development team could work on them if requested.

To protect against disclosure the number of observations in each summarizing group in each source is calculated and if any of these falls below the value of nfilter.tab (the minimum allowable non-zero count in a contingency table) the tapply analysis of that source will return only an error message. The value of nfilter.tab is can be set and modified only by the data custodian. If an analytic team wishes the value to be reduced (e.g. to 1 which will allow any output from tapply to be returned) this needs to formally be discussed and agreed with the data custodian.

If the reason for the tapply analysis is, for example, to break a dataset down into a small number of values for each individual and then to flag up which individuals have got at least one positive value for a binary outcome variable, then that flagging does not have to be overtly returned to the client-side. Rather, it can be written as a vector to the server-side at each source (which, like any other server-side object, cannot then be seen, abstracted or copied). This can be done using ds.tapply.assign which writes the results as a newobj to the server-side and does not test the number of observations in each group against nfilter.tab. For more information see the help option of ds.tapply.assign function.

The native R tapply function has optional arguments such as na.rm = TRUE for FUN = mean which will exclude any NAs from the outcome variable to be summarized. However, in order to keep DataSHIELD's ds.tapply and ds.tapply.assign functions straightforward, the server-side functions tapplyDS and tapplyDS.assign both starts by stripping any observations which have missing (NA) values in either the outcome variable or in any one of the indexing factors. In consequence, the resultant analyses are always based on complete cases.

In INDEX.names argument the native R tapply function can coerce non-factor vectors into factors. However, this does not always work when using the DataSHIELD ds.tapply or ds.tapply.assign functions so if you are concerned that an indexing vector is not being treated correctly as a factor, please first declare it explicitly as a factor using ds.asFactor.

In FUN.name argument the allowable functions are: N or length (the number of (non-missing) observations in the group defined by each combination of indexing factors); mean; SD (standard deviation); sum; quantile (with quantile probabilities set at c(0.05,0.1,0.2,0.25,0.3,0.33,0.4,0.5,0.6,0.67,0.7,0.75,0.8,0.9,0.95)).

Server function called: ds.tapply.assign

Value

ds.tapply.assign returns an array of the summarized values. The array is written to the server-side. It has the same number of dimensions as INDEX.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Apply a Function Over a Server-Side Ragged Array.
  # Write the resultant object on the server-side


  ds.assign(toAssign = "D$LAB_TSC",
            newobj = "LAB_TSC",
            datasources = connections)

  ds.assign(toAssign = "D$GENDER",
            newobj =  "GENDER",
            datasources = connections)

  ds.tapply.assign(X.name = "LAB_TSC",
                   INDEX.names = c("GENDER"),
                   FUN.name = "mean",
                   newobj="fun_mean.newobj",
                   datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

ds.testObjExists *Checks if an R object exists on the server-side*

Description

This function checks that a specified data object exists or has been correctly created on a specified set of data servers.

Usage

```
ds.testObjExists(test.obj.name = NULL, datasources = NULL)
```

Arguments

test.obj.name	a character string specifying the name of the object to search.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Close copies of the code in this function are embedded into other functions that create an object and you then wish to test whether it has successfully been created e.g. ds.make or ds.asFactor.

Server function called: testObjExistsDS

Value

ds.testObjExists returns a list of messages specifying that the object exists on the server-side. If the specified object does not exist in at least one of the specified data sources or it exists but is of class NULL, the function returns an error message specifying that the object does not exist in all data sources.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
```

```
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Check if D object exists on the server-side

  ds.testObjExists(test.obj.name = "D",
                   datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)


## End(Not run)
```

---

ds.unique *Perform 'unique' on a variable on the server-side*

Description

Perform 'unique', from the 'base' package on a specified variable on the server-side

Usage

```
ds.unique(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x.name	a character string providing the name of the variable, in the server, to perform unique upon
newobj	a character string that provides the name for the output object that is stored on the data servers. Default unique.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

Will create a vector or list which has no duplicate values.

Server function called: uniqueDS

Value

ds.unique returns the vector of unique R objects which are written to the server-side.

Author(s)

Stuart Wheater, DataSHIELD Development Team

Examples

```
## Not run:
  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  # Create a vector with combined objects
  ds.unique(x.name = "D$LAB_TSC", newobj = "new.vect", datasources = connections)

  # Clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

---

ds.unList *Flattens Server-Side Lists*

Description

Coerces an object of list class back to the class it was when it was coerced into a list.

Usage

```
ds.unList(x.name = NULL, newobj = NULL, datasources = NULL)
```

Arguments

x.name	a character string specifying the name of the input object to be unlisted.
newobj	a character string that provides the name for the output variable that is stored on the data servers. Default unlist.newobj.
datasources	a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the native R function unlist.

When an object is coerced to a list, depending on the class of the original object some information may be lost. Thus, for example, when a data frame is coerced to list the information that underpins the structure of the data frame is lost and when it is subject to the function ds.unList it is returned to a simpler class than data frame e.g. numeric (basically a numeric vector containing all of the original data in all variables in the data frame but with no structure). If you wish to reconstruct the original data frame you, therefore, need to specify this structure again e.g. the column names, etc.

Server function called: unListDS

Value

ds.unList returns to the server-side the unlist object. Also, two validity messages are returned to the client-side indicating whether the new object has been created in each data source and if so whether it is in a valid form.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:
  ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
```

```
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM2", driver = "OpalDriver")
builder$append(server = "study3",
                url = "http://192.168.56.100:8080/",
                user = "administrator", password = "datashield_test&",
                table = "CNSIM.CNSIM3", driver = "OpalDriver")
logindata <- builder$build()

connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

#Create a list on the server-side

ds.asList(x.name = "D",
          newobj = "list.D",
          datasources = connections)

#Flatten a server-side lists

ds.unList(x.name = "list.D",
          newobj = "un.list.D",
         datasources = connections)

# Clear the Datashield R sessions and logout
datashield.logout(connections)


## End(Not run)
```

---

ds.var *Computes server-side vector variance*

Description

Computes the variance of a given server-side vector.

Usage

```
ds.var(x = NULL, type = "split", checks = FALSE, datasources = NULL)
```

Arguments

x	a character specifying the name of a numerical vector.
type	a character string that represents the type of analysis to carry out. This can be set as 'combine', 'combined', 'combines', 'split', 'splits', 's', 'both' or 'b'. For more information see **Details**.
checks	logical. If TRUE optional checks of model components will be undertaken. Default is FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed.

datasources — a list of DSConnection-class objects obtained after login. If the datasources argument is not specified the default set of connections will be used: see datashield.connections_defa

Details

This function is similar to the R function var.

The function can carry out 3 types of analysis depending on the argument type:
(1) If type is set to 'combine', 'combined', 'combines' or 'c', a global variance is calculated.
(2) If type is set to 'split', 'splits' or 's', the variance is calculated separately for each study.
(3) If type is set to 'both' or 'b', both sets of outputs are produced.

Server function called: varDS

Value

ds.var returns to the client-side a list including:

Variance.by.Study: estimated variance, Nmissing (number of missing observations), Nvalid (number of valid observations) and Ntotal (sum of missing and valid observations) separately for each study (if type = split or type = both).
Global.Variance: estimated variance, Nmissing, Nvalid and Ntotal across all studies combined (if type = combine or type = both).
Nstudies: number of studies being analysed.
ValidityMessage: indicates if the analysis was possible.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

 ## Version 6, for version 5 see the Wiki

  # connecting to the Opal servers

  require('DSI')
  require('DSOpal')
  require('dsBaseClient')

  builder <- DSI::newDSLoginBuilder()
  builder$append(server = "study1",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM1", driver = "OpalDriver")
  builder$append(server = "study2",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM2", driver = "OpalDriver")
```

```
  builder$append(server = "study3",
                 url = "http://192.168.56.100:8080/",
                 user = "administrator", password = "datashield_test&",
                 table = "CNSIM.CNSIM3", driver = "OpalDriver")
  logindata <- builder$build()

  connections <- DSI::datashield.login(logins = logindata, assign = TRUE, symbol = "D")

  #Calculate the variance of a vector in the server-side

  ds.var(x = "D$LAB_TSC",
         type = "split",
         checks = FALSE,
         datasources = connections)

  # clear the Datashield R sessions and logout
  datashield.logout(connections)

## End(Not run)
```

`ds.vectorCalc` *Performs a mathematical operation on two or more vectors*

Description

Carries out a row-wise operation on two or more vector. The function calls no server side function; it uses the R operation symbols built in DataSHIELD.

Usage

```
ds.vectorCalc(x = NULL, calc = NULL, newobj = NULL, datasources = NULL)
```

Arguments

`x`	a vector of characters, the names of the vectors to include in the operation.
`calc`	a character, a symbol that indicates the mathematical operation to carry out: '+' for addition, '/' for division, *' for multiplication and '-' for subtraction.
`newobj`	the name of the output object. By default the name is 'vectorcalc.newobj'.
`datasources`	a list of `DSConnection-class` objects obtained after login. If the <datasources> the default set of connections will be used: see datashield.connections_default.

Details

In DataSHIELD it is possible to perform an operation on vectors by just using the relevant R symbols (e.g. '+' for addition, '*' for multiplication, '-' for subtraction and '/' for division). This might however be inconvenient if the number of vectors to include in the operation is large. This function

takes the names of two or more vectors and performs the desired operation which could be an addition, a multiplication, a subtraction or a division. If one or more vectors have a missing value at any one entry (i.e. observation), the operation returns a missing value ('NA') for that entry; the output vectors has, hence the same length as the input vectors.

Value

no data are returned to user, the output vector is stored on the server side.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load the file that contains the login details
  data(logindata)

  # login and assign the required variables to R
  myvar <- list('LAB_TSC','LAB_HDL')
  conns <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # performs an addtion of 'LAB_TSC' and 'LAB_HDL'
  myvectors <- c('D$LAB_TSC', 'D$LAB_HDL')
  ds.vectorCalc(x=myvectors, calc='+')

  # clear the Datashield R sessions and logout
  datashield.logout(conns)


## End(Not run)
```

Index