Package 'dsBaseClient'

September 5, 2019

Title DataSHIELD Client Functions

Version 5.0.0

Author DataSHIELD Developers <datashield@newcastle.ac.uk>

Maintainer DataSHIELD Developers <datashield@newcastle.ac.uk>

Description DataSHIELD client functions for the client site.

License GPL-3

Depends R (>= 3.5.0)

Imports fields,
metafor,
opaladmin,
opal

RoxygenNote 6.1.1

Encoding UTF-8

R topics documented:

`ds.asCharacter` *ds.asCharacter calling assign function asCharacterDS*

Description

this function is based on the native R function as.character

Usage

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

Arguments

`x.name`	the name of the input object to be coerced to class character. Must be specified in inverted commas.
`newobj`	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.char
`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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

See details of the native R function as.character.

Value

the object specified by the <newobj> argument (or by default <x.name>.char if the <newobj> argument is NULL) 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

`ds.asDataMatrix` *ds.asDataMatrix calling assign function asDataMatrixDS*

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`	the name of the input object to be coerced to a data.matrix. Must be specified in inverted commas eg x.name="name.of.object"
`newobj`	the name of the new output variable specified in inverted commas. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.mat
`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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

This function is based on the native R function data.matrix If applied to a data.frame, the native R function as.matrix converts all columns into character class. In contrast, if applied to a data.frame the native R function data.matrix converts the data.frame to a matrix but maintains all data columns in their original class

Value

the object specified by the <newobj> argument (or by default <x.name>.mat if the <newobj> argument is NULL) 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton, for DataSHIELD Development Team

---

ds.asFactor *Converts a numeric vector into a factor type*

Description

This function assigns a numeric vector into a factor type

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	the name of the variable that is to be converted to a factor
newobj.name	the name of the new object. If this argument is set to NULL or not specified then the default name of the new variable is the name of the input variable with the suffixe '.f'.

`forced.factor.levels`

the levels that the user wants to split the input variable. If this argument is set to NULL (default) then a vector with all unique levels from all studies is created as a result of the output of 'asFactorDS1.b' server-side function. The 'asFactorDS1.b' function returns the levels of the input variable from each single study in ascending order if the levels are numeric or in alphabetic order if the levels are characters. Then the levels from each study are combined together and a vector with all unique levels is created.

`fixed.dummy.vars`

a boolean that determines whether the new object is represented as a vector or as a matrix with elements dummy variables indicating the factor level of each data point. If this argument is set to FALSE (default) then the input variable is converted to a factor and assigned as a vector. If is set to TRUE then the input variable is converted to a factor but presented as a matrix of dummy variables. The matrix of dummy variables also depends on argument `baseline.level`. To understand how this matrix 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 arqyment `fixed.dummy.vars` to TRUE and the `baseline.level` to 1 which is the default value and the `forced.factor.levels` equal to c(1,2,3,4,5) then the input vector is converted to the following matrix of dummy variables: DV2 DV3 DV4 DV5 [1,] 0 0 0 0 [2,] 1 0 0 0 [3,] 0 0 0 0 [4,] 0 1 0 0 [5,] 0 0 1 0 [6,] 0 0 1 0 [7,] 0 0 0 0 [8,] 0 1 0 0 [9,] 0 0 1 0 [10,] 0 0 0 1 For the same example if the `baseline.level` is set to be equal to 3 then the matrix is: DV1 DV2 DV4 DV5 [1,] 1 0 0 0 [2,] 0 1 0 0 [3,] 1 0 0 0 [4,] 0 0 0 0 [5,] 0 0 1 0 [6,] 0 0 1 0 [7,] 1 0 0 0 [8,] 0 0 0 0 [9,] 0 0 1 0 [10,] 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 column (column 'DV2') 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 column (column 'DV1') 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).

`baseline.level`

a number 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 baseline level is not taken into account. If the `fixed.dummy.vars` is set to TRUE then the baseline level is used as explained above. 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 are. 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 (let's say for example `baseline.level=8`) the the matrix of dummy variables is: DV1 DV2 DV3 DV4 DV5 [1,] 1 0 0 0 0 [2,] 0 1 0 0 0 [3,] 1 0 0 0 0 [4,] 0 0 1 0 0 [5,] 0 0 0 1 0 [6,] 0 0 0 1 0 [7,] 1 0 0 0 0 [8,] 0 0 1 0 0 [9,] 0 0 0 1 0 [10,] 0 0 0 0 1 The creation of a factor in the form of a matrix with dummy variables at different baseline levels is useful

in survival analysis for example in piecewise exponential regression where the baseline hazard is different in different time intervals. For more details see the description of ds.glm function.

`datasources` a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources. By default an internal function looks for 'opal' objects in the environment and sets this parameter.

Details

Converts a numeric vector into a factor type which is represented either as a vector or as a matrix of dummy variables dependending on the argument `fixed.dummy.vars`.

Value

all the unique levels of the converted variable and the tracer of the function

Examples

```
## Not run:

#  # load that contains the login details
#  logindata.VMs.em <- ds.createLogindata(110,110,110,table=c("SURVIVAL.EXPAND_WITH_MISSING1",
#                       "SURVIVAL.EXPAND_WITH_MISSING2","SURVIVAL.EXPAND_WITH_MISSING3"))
#  logindata.VMs.em <- logindata.VMs.em[1:3,]
#  opals.em <- opal::datashield.login(logins=logindata.VMs.em,assign=TRUE,symbol="EM")
#
#  ds.asNumeric("EM$time.id","TID")
#
#  Example 1
#  ds.asFactor("TID","TID.f")
#  ds.class("TID.f")
#  ds.table1D("TID.f")
#
#  Example 2
#  ds.asFactor("TID","TID.f2",forced.factor.levels=1:6)
#  ds.class("TID.f2")
#  ds.table1D("TID.f2")
#
#  Example 3
#  ds.asFactor("TID","TID.f3",forced.factor.levels=0:10)
#  ds.class("TID.f3")
#  ds.table1D("TID.f3")
#
#  Example 4
#  ds.asFactor("TID","TID.f4",forced.factor.levels=2:3)
#  ds.class("TID.f4")
#  ds.table1D("TID.f4")
#
#  Example 5
#  ds.asFactor("TID","TID.f5",forced.factor.levels=c(1,2,3,4,"a","h",5))
#  ds.class("TID.f5")
```

```
#  ds.table1D("TID.f5")
#
#  Example 6
#  ds.asFactor("TID","TID.mat1",fixed.dummy.vars=TRUE)
#  ds.class("TID.mat1")
#
#  Example 7
#  ds.asFactor("TID","TID.mat6",fixed.dummy.vars=TRUE,baseline.level=6)
#  ds.class("TID.mat6")


## End(Not run)
```

`ds.asInteger` *ds.asInteger calling assign function asIntegerDS*

Description

this function is based on the native R function as.integer

Usage

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

Arguments

`x.name`	the name of the input object to be coerced to class integer. Must be specified in inverted commas.
`newobj`	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.int
`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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

See details of the native R function as.integer.

Value

the object specified by the <newobj> argument (or by default <x.name>.int if the <newobj> argument is NULL) 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

`ds.asList` *ds.asList calling aggregate function asListDS*

Description

this function is based on the native R function as.list

Usage

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

Arguments

`x.name`	the name of the input object to be coerced to class list. Must be specified in inverted commas e.g. x.name="input.object.name"
`newobj`	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.list
`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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

See details of the native R function as.list. Unlike most other class coercing functions the serverside function that is called is an aggregate function rather than an assign function. This is because the datashield.assign function in opal deals specially with a created object (<newobj>) if it is of class list. Reconfiguring the function as an aggregate function works around this problem.

Value

the object specified by the <newobj> argument (or by default <x.name>.list if the <newobj> argument is NULL) which is written to the serverside. In addition, two validity messages are returned. The first confirms an output object has been created, the second states its class. The way that as.list coerces objects to list depends on the class of the object, but in general the class of the output object should usually be 'list'

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

---

ds.asLogical *ds.asLogical calling assign function asLogicalDS*

Description

this function is based on the native R function as.logical

Usage

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

Arguments

x.name	the name of the input object to be coerced to class logical. Must be specified in inverted commas.
newobj	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.logic
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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

See details of the native R function as.logical.

Value

the object specified by the <newobj> argument (or by default <x.name>.logic if the <newobj> argument is NULL) 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton, for DataSHIELD Development Team

---

ds.asMatrix *ds.asMatrix calling assign function asMatrixDS*

Description

this function is based on the native R function as.matrix

Usage

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

Arguments

x.name	the name of the input object to be coerced to a matrix. Must be specified in inverted commas.
newobj	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.mat
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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

This function is based on the native R function as.matrix If applied to a data.frame, the native R function as.matrix converts all columns into 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 native R function data.matrix and in DataSHIELD this is called by ds.asDataMatrix which calls asDataMatrixDS

Value

the object specified by the <newobj> argument (or by default <x.name>.mat if the <newobj> argument is NULL) 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

---

ds.asNumeric *ds.asNumeric calling assign function asNumericDS*

Description

this function is based on the native R function as.numeric

Usage

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

Arguments

x.name	the name of the input object to be coerced to class numeric. Must be specified in inverted commas.
newobj	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.num
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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

See details of the native R function as.numeric.

Value

the object specified by the <newobj> argument (or by default <x.name>.num if the <newobj> argument is NULL) 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

---

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

Description

This function assigns a datashield object to a name, hence creating a new object. It also calls 'assign' server side functions to generate objects stored on the server side. The function is a wrapper for the 'opal' package function 'datashield.assign'.

Usage

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

Arguments

`toAssign`	a string character, the object to assign or the call to an assign function that generates the object to assign.
`newobj`	the name of the new object
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

The new object is stored on the remote R instance (i.e. on the server side). If no name is provided, the new object is named 'newObject', by default.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  # (by default the assigned dataset is a dataframe named 'D')
  myvar <- list("LAB_TSC")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Example 1: assign the variable 'LAB_TSC' in the dataframe D
  ds.assign(toAssign='D$LAB_TSC', newobj='labtsc')

  # Example2: get the log values of the variable 'LAB_TSC' in D and assign it to 'logTSC'
  ds.assign(toAssign='log(D$LAB_TSC)', newobj='logTSC')

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


## End(Not run)
```

`ds.Boole` *ds.Boole*

Description

Converts the individual elements of a vector or other object into Boolean indicators(TRUE/FALSE or 1/0) based on the standard set of Boolean operators: ==, !=, >, >=, <, <=.

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 or scalar to which <V1> is to be compared. So, if <V2> is a scalar (e.g. '4') and the Boolean operator is '<=', the output vector will be a binary/Boolean variable with elements taking the value 1 or TRUE if the corresponding element of <V1> is 4 or less and 0 or FALSE otherwise. On the other hand, if <V2> is a vector and the Boolean operator is '==', the output vector will be a binary/Boolean variable with elements taking the value 1 or TRUE if the corresponding elements of <V1> and <V2> are equal and 0 or FALSE otherwise. If <V2> is a vector rather than a scalar it must be of the same length as <V1>
Boolean.operator	A character string specifying one of six possible Boolean operators: '==', '!=', '>', '>=', '<', '<='
numeric.output	a TRUE/FALSE indicator defaulting to TRUE determining whether the final output variable should be of class numeric (1/0) or class logical (TRUE/FALSE). It is easy to convert a logical class variable to numeric using the ds.asNumeric() function and to convert a numeric (1/0) variable to logical you can apply ds.Boole with <Boolean.operator> '==', <V2> the scalar '1' and <numeric.output> FALSE.
na.assign	A character string taking values 'NA', '1' or '0'. If 'NA' then any NA values in the input vector remain as NAs in the output vector. If '1' or '0' NA values in the input vector are all converted to 1 or 0 respectively.
newobj	A character string specifying the name of the vector to which the output vector is to be written. If no <newobj> argument is specified, the output vector defaults to "V1_Boole" where <V1> is the first argument of the function.
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)]

Details

A combination of operators reflected in AND can be obtained by multiplying two or more binary/Boolean vectors together: observations taking the value 1 in every vector will then take the value 1 while all others will take the value 0. The combination OR can be obtained by adding two or more vectors and then then reapply ds.Boole using the operator >= 1: any observation taking the value 1 in one or more vectors will take the value 1 in the final vector.

Value

the object specified by the <newobj> argument (or default name <V1>_Boole) 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.Boole also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

DataSHIELD Development Team

`ds.c` *Combines values into a vector or list*

Description

Concatenates object into one object.

Usage

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

Arguments

`x`	a character, a vector that holds the names of the objects to combine.
`newobj`	the name of the output object. If this argument is set to NULL, the name of the new object is 'newObject'.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

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

Value

nothing is returned to the client, the new object 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 specific variable(s)
  # (by default the assigned dataset is a dataframe named 'D')
  myvar <- c('LAB_TSC', 'LAB_HDL')
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Get the variables 'LAB_TSC' by 'LAB_HDL' from the dataframe 'D' and combine them
  myvect <- c('D$LAB_TSC', 'D$LAB_HDL')
  ds.assign(toAssign='D$LAB_TSC', newobj='labtsc')
  ds.assign(toAssign='D$LAB_HDL', newobj='labhdl')
  myvect <- c('labtsc', 'labhdl')
  ds.c(x=myvect)

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


## End(Not run)
```

---

ds.cbind *ds.cbind calling cbindDS*

Description

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

Usage

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

Arguments

x	This is a vector of character strings representing the names of the elemental components to be combined For example, the call: ds.cbind(x=c('DF_input','matrix.m','var_age'),newobj='cb will combine a pre-existing data.frame called DF_input with a matrix and a variable called var_age. The output will be the object cbind_output in which the first columns will be the columns of DF_input, to their right the next block of columns are from matrix.m and the final column will be the variable var_age. As many elemental components as needed may be combined in any order e.g. 3

data.frames, 7 variables and 2 matrices. For convenience the x argument can alternatively be specified in a two step procedure, the first being a call to the native R environment on the client server: x.components<-c('DF_input1','matrix.m','DF_input2', 'var_age'); ds.cbind(x=x.components,newobj='DF_output'). In order to disambiguate column names, if the same column name appears several times the suffix '.1' will be appended to the second instance, '.2' to the third and, generally, .(n-1) to the nth instance. Disambiguation does not occur if column names are user specified using <force.colnames>

`DataSHIELD.checks`
: logical, if TRUE checks are made that all input objects exist and are of an appropriate class. These checks are relatively slow and so the <DataSHIELD.checks> argument is defaulted to FALSE

`force.colnames`
: NULL or a vector of character strings representing the required column names of the output object. For example: force.colnames=c("colname1","name.of.second.column", "lastcol") for an output object with three columns. If <force.colnames> is NULL column names are inferred from the names or column names of the input objects - please see 'details' for disambiguation. If <force.colnames> is not NULL, there is no disambiguation so you can force columns to have the same names should you so wish.The vector of column names must have the same number of elements as there are columns in the output object. If the length of the column name vector is incorrect a studysideMessage is returned: "Number of column names does not match number of columns in output object. Here 'N' names are required.Please see help for ds.cbind function" where 'N' is the actual number of columns in the output object

`newobj`
: This a character string providing a name for the output data.frame which defaults to 'cbind.out' 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

A sequence of vector, matrix or data-frame arguments is combined column by column to produce a matrix written to the which is written to the serverside. For more details see the native R function cbind. The handling of argument <x> is the same as for ds.dataFrame

Value

the object specified by the <newobj> argument (or default name <cbind.out>). which is written to the serverside. Just like the cbind function in native R, the output object is of class matrix unless one

or more of the input objects is a data.frame in which case the class of the output object is data.frame. In the latter case, if an object of class matrix is required one may use the ds.asMatrix function. As well as writing the output object as <newobj> on the serverside, 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.cbind() also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("<newobj>") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("<newobj>") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.changeRefGroup` *Changes the reference level of a factor*

Description

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, the name of a vector of type factor.
`ref`	the reference level
`newobj`	the name of the new variable. If this argument is set to NULL, the name of the new variable is the name of the input variable with the suffixe '_newref'.
`reorderByRef`	a boolean that tells whether or not the new vector should be ordered by the reference group (i.e. putting the reference group first). The default is to not re-order for the reasons explained in the 'details' section.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

In addition to what the R function does, this function allows for the user to re-order the vector, putting the reference group first. 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.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Isaeva, J.; Gaye, A.

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 matrix or a data frame.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the stored variables
  # (by default the assigned dataset is a dataframe named 'D')
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # 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')

  # 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')

  # print out the levels of the new vector
  ds.levels(x='bmi_new')

  # by default the reference is the first level in the vector of levels (here 'normal')
  # now change and set the reference to 'obesity' without changing the order (default)
  ds.changeRefGroup(x='bmi_new', ref='obesity', newobj='bmi_ob')

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

  # Example 2: change the reference and re-order by the refence 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')
  ds.changeRefGroup(x='bmi_new', ref='obesity', newobj='bmi_ob', reorderByRef=TRUE)

  # clear the Datashield R sessions and logout
```

```
  datashield.logout(opals)

## End(Not run)
```

`ds.class` *Retrieves the class of an object*

Description

This function is similar to R function `class`.

Usage

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

Arguments

`x`	an R object
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

Same as for the R function `class`.

Value

a character the type of x

Author(s)

Gaye, A.; Isaeva, J.

See Also

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

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the stored variables
  # (by default the assigned dataset is a dataframe named 'D')
  opals <- datashield.login(logins=logindata,assign=TRUE)
```

```
  # Example 1: Get the class of the whole dataset
  ds.class(x='D')

  # Example 2: Get the class of the variable PM_BMI_CONTINUOUS
  ds.class(x='D$LAB_TSC')

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


## End(Not run)
```

`ds.colnames` *Retrieves column names of a matrix-like object*

Description

this function is similar to R function `colnames`.

Usage

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

Arguments

`x`	a character, the name of a dataframe or matrix.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

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

Value

a character vector

Author(s)

Gaye, A.; Isaeva, J.

See Also

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

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the stored variables
  # (by default the assigned dataset is a dataframe named 'D')
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # Get the column names of the assigned datasets (default name is 'D')
  ds.colnames(x='D')

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


## End(Not run)
```

`ds.contourPlot` *Generates a contour plot*

Description

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

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, the name of a numerical vector.
`y`	a character, the name of a numerical vector.
`type`	a character which 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 which 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`	a number of intervals for a density grid object.

method	a character which defines which contour will be created. If method is set to 'smallCellsRule' (default option), 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 neighbours 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. 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.
k	the number of the nearest neghbours for which their centroid is calculated. 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'.
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'. 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.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

The 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 minumum and maximum values but rather close approximates of the real minimum and maximum value. This was done to reduce the risk of potential disclosure.

Value

a contour plot

Author(s)

Julia Isaeva, Amadou Gaye, Paul Burton, Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # load the file that contains the login details
  data(logindata)
```

```
# login and assign specific variables(s)
# (by default the assigned dataset is a dataframe named 'D')
opals <- datashield.login(logins=logindata,assign=TRUE)

# Example 1: Plot a combined (default behaviour) contour plot of the variables 'LAB_TSC'
# and 'LAB_HDL' using the method 'LowCountsRule' (default method) that applies a stochastic noise
# in the extreme values of the variables' range.
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL')

# Example 2: the same as example 1
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="smallCellsRule", type='combine')

# Example 3: similar as example 2 but for type='split'
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="smallCellsRule", type='split')

# Example 4: Plot a combined (default behaviour) contour plot of the variables 'LAB_TSC'
# and 'LAB_HDL' using the method 'deterministic' that plots the exact contour plot of the
# centroids of each 3 (default number) nearest neighbours.
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic")

# Example 5: the same as example 4
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic", k=3, type='combine')

# Example 6: similar as example 5 for type='split'
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic", k=3, type='split')

# Example 7: similar as example 6 for k=7
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic", k=7, type='split')

# Example 8: similar as example 7 for numints=40
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', numints=40, method="deterministic", k=7,
               type='split')

# Example 9: Plot a combined (default behaviour) contour plot of the variables 'LAB_TSC'
# and 'LAB_HDL' using the method 'probabilistic' that plots the exact contour plot of the
# noisy data
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic")

# Example 10: the same as example 9
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic", noise=0.25, type='combine')

# Example 11: the same as example 10 but for bigger level of noise
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic", noise=2, type='combine')

# Example 12: the same as example 11 but for type='split'
ds.contourPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic", noise=2, type='split')

# Example 13: if any of the input variables is a factor then the function fails
ds.contourPlot(x='D$LAB_TSC', y='D$GENDER')

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

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

`ds.cor` *Calculates the correlation between two variables*

Description

This function calculates the correlation of two variables or the correlation matrix for the vairables of an input dataframe

Usage

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

Arguments

`x`	a character, the name of a vector, matrix or dataframe of variable(s) for which the correlation(s) is (are) calculated for.
`y`	NULL (default) or the name of a vector, matrix or dataframe with compatible dimensions to x.
`naAction`	a character string giving a method for computing correlations in the presence of missing values. This must be one of the strings "casewise.complete" or "pairwise.complete". If `use` is set to 'casewise.complete', then the function omits all the rows in the whole dataframe that include at least one cell with a missing value before the calculation of correlations. If `use` is set to 'pairwise.complete' (default), then the function divides the input dataframe to subset subset dataframes 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 correlations of those pairs.
`type`	a character which represents the type of analysis to carry out. If `type` is set to 'split' (default), the correlation of two variables or the variance-correlation matrix of an input dataframe and the number of complete cases and missing values are returned for each 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.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

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 for the user to make a decision about the 'relevance' of the correlation based on the number of complete cases included in the correlation calculations.

Value

a list containing the number of missing values in each variable, the number of missing variables casewise or paiwise depending on the argument use, the correlation matrix, the number of used complete cases and an error message which indicates whether or not the input variables pass the disclosure control (i.e. none of them is dichotomous with a level having less counts than the pre-specified threshold). If any of the input variables does not pass the disclosure control then all the output values are replaced with NAs. If all the variables are valid and pass the control, then the output matrices are returned and also an error message is returned but it is replaced by NA.

Author(s)

Gaye A; Avraam D; Burton PR

Examples

```
## Not run:

#  # load that contains the login details
#  data(glmLoginData)
#  library(opal)
#
#  # login and assign specific variable(s)
#  # (by default the assigned dataset is a dataframe named 'D')
#  myvar <- list('LAB_HDL', 'LAB_TSC', 'LAB_GLUC_ADJUSTED', 'GENDER')
#  opals <- opal::datashield.login(logins=glmLoginData, assign=TRUE, variables=myvar)
#
#  # Example 1: generate the correlation matrix for the assigned dataset 'D'
#  # which contains 4 vectors (3 continuous and 1 categorical)
#  ds.cor(x='D')
#
#  # Example 2: generate the correlation matrix for the dataset 'D' combined for all
#  # studies and removing any missing values casewise
#  ds.cor(x='D', naAction='casewise.complete', type='combine')
#
#  # Example 3: calculate the correlation between two vectors
#  # (first assign the vectors from 'D')
#  ds.assign(newobj='labhdl', toAssign='D$LAB_HDL')
#  ds.assign(newobj='labtsc', toAssign='D$LAB_TSC')
#  ds.assign(newobj='gender', toAssign='D$GENDER')
#  ds.cor(x='labhdl', y='labtsc', naAction='pairwise.complete', type='combine')
#  ds.cor(x='labhdl', y='labtsc', naAction='casewise.complete', type='combine')
#  ds.cor(x='labhdl', y='gender', naAction='pairwise.complete', type='combine')
#  ds.cor(x='labhdl', y='gender', naAction='casewise.complete', type='combine')
#
#  # clear the Datashield R sessions and logout
#  opal::datashield.logout(opals)


## End(Not run)
```

ds.corTest *Tests for correlation between paired samples*

Description

This is similar to the R base function 'cor.test'.

Usage

```
ds.corTest(x = NULL, y = NULL, datasources = NULL)
```

Arguments

x	a character, the name of a numerical vector
y	a character, the name of a numerical vector
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

Runs a two sided pearson test with a 0.95 confidence level.

Value

a list containing the results of the test

Author(s)

Gaye, A.; Burton, P.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  # (by default the assigned dataset is a dataframe named 'D')
  myvar <- list('LAB_TSC', 'LAB_HDL')
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # test for correlation between the variables 'LAB_TSC' and 'LAB_HDL'
  ds.corTest(x='D$LAB_TSC', y='D$LAB_HDL')

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

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

---

ds.cov *Calculates the covariance between two variables*

Description

This function calculates the covariance of two variables or the variance-covariance matrix for the vairables of an input dataframe

Usage

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

Arguments

x	a character, the name of a vector, matrix or dataframe of variable(s) for which the covariance(s) is (are) calculated for.
y	NULL (default) or the name of a vector, matrix or dataframe with compatible dimensions to x.
naAction	a character string giving a method for computing covariances in the presence of missing values. This must be one of the strings "casewise.complete" or "pairwise.complete". If use is set to 'casewise.complete', then the function omits all the rows in the whole dataframe that include at least one cell with a missing value before the calculation of covariances. If use is set to 'pairwise.complete' (default), then the function divides the input dataframe to subset subset dataframes 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.
type	a character which represents the type of analysis to carry out. If type is set to 'split' (default), the covariance of two variables or the variance-covariance matrix of an input dataframe and the number of complete cases and missing values are returned for each 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.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

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 make a decision about the 'relevance' of the covariance based on the number of complete cases included in the covariance calculations.

Value

a list containing the number of missing values in each variable, the number of missing variables casewise or paiwise depending on the argument use, the covariance matrix, the number of used complete cases and an error message which indicates whether or not the input variables pass the disclosure control (i.e. none of them is dichotomous with a level having less counts than the pre-specified threshold). If any of the input variables does not pass the disclosure control then all the output values are replaced with NAs. If all the variables are valid and pass the control, then the output matrices are returned and also an error message is returned but it is replaced by NA.

Author(s)

Gaye A; Avraam D; Burton PR

Examples

```
## Not run:

#  # load that contains the login details
#  data(glmLoginData)
#  library(opal)
#
#  # login and assign specific variable(s)
#  # (by default the assigned dataset is a dataframe named 'D')
#  myvar <- list('LAB_HDL', 'LAB_TSC', 'LAB_GLUC_ADJUSTED', 'GENDER')
#  opals <- opal::datashield.login(logins=glmLoginData, assign=TRUE, variables=myvar)
#
#  # Example 1: generate the covariance matrix for the assigned dataset 'D'
#  # which contains 4 vectors (3 continuous and 1 categorical)
#  ds.cov(x='D')
#
#  # Example 2: generate the covariance matrix for the dataset 'D' combined for all
#  # studies and removing any missing values casewise
#  ds.cov(x='D', naAction='casewise.complete', type='combine')
#
#  # Example 3: calculate the covariance between two vectors
#  # (first assign the vectors from 'D')
#  ds.assign(newobj='labhdl', toAssign='D$LAB_HDL')
#  ds.assign(newobj='labtsc', toAssign='D$LAB_TSC')
#  ds.assign(newobj='gender', toAssign='D$GENDER')
#  ds.cov(x='labhdl', y='labtsc', naAction='pairwise.complete', type='combine')
#  ds.cov(x='labhdl', y='labtsc', naAction='casewise.complete', type='combine')
#  ds.cov(x='labhdl', y='gender', naAction='pairwise.complete', type='combine')
#  ds.cov(x='labhdl', y='gender', naAction='casewise.complete', type='combine')
#
#  # clear the Datashield R sessions and logout
#  opal::datashield.logout(opals)


## End(Not run)
```

ds.dataFrame *ds.dataFrame calling dataFrameDS*

Description

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

Usage

```
ds.dataFrame(x = NULL, row.names = NULL, check.rows = FALSE,
  check.names = TRUE, stringsAsFactors = TRUE, completeCases = FALSE,
  DataSHIELD.checks = FALSE, newobj = "df_new", datasources = NULL,
  notify.of.progress = FALSE)
```

Arguments

x	This is a vector of character strings representing the names of the elemental components to be combined. For example, the call: ds.dataFrame(x=c('DF_input','matrix.m','var_age'),newo will combine a pre-existing data.frame called DF_input with a matrix and a variable called var_age. The output will be the combined data.frame DF_output. As many elemental components as needed may be combined in any order e.g. 3 data.frames, 7 variables and 2 matrices. For convenience the x argument can alternatively be specified in a two step procedure, the first being a call to the native R environment on the client server: x.components<-c('DF_input1','matrix.m','DF_input2', 'var_age'); ds.dataFrame(x=x.components,newobj='DF_output')
row.names	NULL or a single integer or character string specifying a column to be used as row names, or a character or integer vector giving the row names for the data frame.
check.rows	if TRUE then the rows are checked for consistency of length and names.
check.names	logical. If TRUE then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted (by make.names) so that they are. As a slight modification to the standard data.frame() function in native R, if any column names are duplicated, the second and subsequent occurances are given the suffixes .1, .2 etc by ds.dataFrame and so there are never any duplicates when check.names is invoked by the serverside function dataFrameDS
stringsAsFactors	logical: should character vectors be converted to factors? The 'factory-fresh' default is TRUE.
completeCases	logical. Default FALSE. If TRUE then any rows with missing values in any of the elemental components of the final output data.frame will be deleted.
DataSHIELD.checks	logical: If TRUE undertakes all DataSHIELD checks (time consuming). Default FALSE.

`newobj` This a character string providing a name for the output data.frame which defaults to 'df_new' 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

A data frame is a list of variables all with the same number of rows with unique row names, which is of class 'data.frame'. ds.dataFrame will create a data frame by combining a series of elemental components which may be pre-existing data.frames, matrices or variables. A critical requirement is that the length of all component variables, and the number of rows of the component data.frames or matrices must all be the same. The output data.frame will then have this same number of rows. ds.dataFrame calls the serverside function dataFrameDS which is almost the same as the native R function data.frame() and so several of the arguments are precisely the same as for data.frame()

Value

the object specified by the <newobj> argument (or default name <df_new>). 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.dataFrame() also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

DataSHIELD Development Team

ds.dataFrameSort *ds.dataFrameSort calling dataFrameSortDS*

Description

Sorts a data frame using a specified sort key

Usage

```
ds.dataFrameSort(df.name = NULL, sort.key.name = NULL,
  sort.descending = FALSE, sort.alphabetic = FALSE,
  sort.numeric = FALSE, newobj = NULL, datasources = NULL)
```

Arguments

df.name	a character string providing the name for 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.alphabetic	logical, if TRUE the sort key is treated as if alphabetic Default=FALSE.
sort.numeric	logical, if TRUE the sort key is treated as if numeric Default=FALSE.
newobj	This a character string providing a name for the output data.frame which defaults to '<df.name>.sorted' if no name is specified where <df.name> is the first argument of ds.dataFrameSort().
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)]

Details

A data frame is a list of variables all with the same number of rows, which is of class 'data.frame'. ds.dataFrameSort will sort a specified data.frame on the serverside using a sort key also on the serverside. The sort key can either sit in the data.frame or outside it. The sort key can be forced to be interpreted as either alphabetic or numeric but not both. If neither interpretation is forced, the sort.key will interpreted naturally: as numeric if it is numeric, otherwise as alphabetic ie as if it is a vector of character strings.

It should be noted that although we are all well used to seeing numbers sorted numerically, and character strings (words) sorted alphabetically when a numeric vector is sorted alphabetically, the order

can look confusing. It is worth mentioning this because Opal sometimes sorts its data tables alphabetically using ID. This can be confusing and one of the reasons for using the ds.dataFrameSort() function is to re-sort a data.frame derived from an Opal data table so its order is more natural. To explain alphabetic and numeric sorting further, here are some illustrations. SORTING NUMBERS: vector.2.sort = (-192 76 841 NA 1670 163 147 101 -112 -231 -9 119 112 NA) numeric.sort = (-231 -192 -112 -9 76 101 112 119 147 163 841 1670 NA NA) alphabetic.sort = (-112 -192 -231 -9 101 112 119 147 163 1670 76 841 NA NA) Notes: Ascending numeric sorting of a numeric vector orders the values in naturally increasing manner with negative values first. The positioning of NAs (missing values) defaults to last. Ascending alphabetic sorting of a numeric vector treats each number as a word and then working from the front of the word, sorts all numbers by the first character of the word, then breaks ties using the second character and so on. ie as is usual in alphabetic sorting of alphabetic words. The sort order can look strange in certain settings. (1) If a number is negative, the first character is '-' which lies ahead of any of the numeric digits in the alphanumeric alphabet. Thus negative numbers appear first (as in numeric sorting) but the ordering of the negative numbers is then the opposite way round to expectation. Thus, instead of the order -231, -192, -112, as in the numeric sort one obtains -112, -192, -231. This is because these three numbers 'tie' on the first character '-' so are then sorted on their second characters. The individual numeric digits have the same alphabetic order as numeric order (0,1,2, ... , 8, 9). This means that having tied on '-' the numbers with second character '1' come before those with second character '2'. (2) Under a numeric sorts, and assuming no decimal point or negative sign, numbers with more digits are of larger magnitude than those with fewer. But under an alphabetic sort all that matters is the sort order of the first character that is not tied with the corresponding number in another number. So the numerically sorted set of numbers 76 841 1670 becomes 1670 76 841 under alphabetic sorting. SORTING CHARACTER STRINGS (WORDS): words.2.sort = ("a" "L" "p" "h" "A" "" "nu" "Me" "R" "IC" "") alphabetic.sort = ("" "" "a" "A" "h" "IC" "L" "Me" "nu" "p" "R") There are few surprises here, perhaps the only thing to note is that missing values (empty strings) get ordered first by default rather than last.

Value

the object specified by the <newobj> argument (or default name <df.name>.sorted) 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 can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

DataSHIELD Development Team

ds.dataFrameSubset *ds.dataFrameSubset calling dataFrameSubsetDS1 and dataFrameSubsetDS2*

Description

Subsets a data frame by row or by column.

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 for the data.frame to be sorted.
V1.name	A character string specifying the name of a subsetting vector to which a Boolean operator will be applied to define the subset to be created. Note if the plan is to subset by column using ALL rows, then <V1.name> might, for example, specify a vector consisting all of ones (see 'details' for how to create such a vector.
V2.name	A character string specifying the name of the vector or scalar to which the values in the vector specified by the argument <V1.name> is to be compared. So, for example, if <V2.name> is a scalar (e.g. '4') and the <Boolean.operator> argument is '<=', the subset data.frame that is created will include all rows that correspond to a value of 4 or less in the subsetting vector specified by the <V1.name> argument. If <V2.name> specifies a vector (which must be of strictly the same length as the vector specified by <V1.name>) and the <Boolean.operator> argument is '==', the subset data.frame that is created will include all rows in which the values in the vectors specified by <V1.name> and <V2.name> are equal. If you are subsetting by column and want to keep all rows in the final subset, <V1.name> can be specified as indicating a "ONES" vector created as described (above) under 'details', <V2.name> can be specified as the scalar "1" and the <Boolean operator> argument can be specified as "=="
Boolean.operator	A character string specifying one of six possible Boolean operators: '==', '!=', '>', '>=', '<', '<='
keep.cols	a numeric vector specifying the numbers of the columns to be kept in the final subset when subsetting by column. For example: keep.cols=c(2:5,7,12) will keep columns 2,3,4,5,7 and 12.
rm.cols	a numeric vector specifying the numbers of the columns to be removed before creating the final subset when subsetting by column. For example: rm.cols=c(2:5,7,12) will remove columns 2,3,4,5,7 and 12.

`keep.NAs` logical, if TRUE any NAs in the vector holding the final Boolean vector indicating whether a given row should be included in the subset will be converted into 1s and so they will be included in the subset. Such NAs could be caused by NAs in either <V1.name> or <V2.name>. If FALSE or NULL NAs in the final Boolean vector will be converted to 0s and the corresponding row will therefore be excluded from the subset.

`newobj` This a character string providing a name for the subset data.frame representing the primary output of the ds.dataFrameSubset() function. This defaults to '<df.name>_subset' if no name is specified where <df.name> is the first argument of ds.dataFrameSubset()

`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

A data frame is a list of variables all with the same number of rows, which is of class 'data.frame'. ds.dataFrameSubset will subset a pre-existing data.frame by specifying the values of a subsetting variable (subsetting by row) or by selecting columns to keep or remove (subsetting by column). When subsetting by row, the resultant subset must strictly be as large or larger than the disclosure trap value nfilter.subset. If you wish to keep all rows in the subset (e.g. if the primary plan is to subset by column not by row) then V1.name 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 NAs. Such a vector can be created as follows: First identify a convenient numeric variable with no missing values (typically a numeric individual ID) let us call it indID, which is equal in length to the data.frame to be subsetted. Then use the ds.make() function with the call ds.make('indID-indID+1','ONES'). This creates a vector of ones (called 'ONES') in each source equal in length to the indID vector in that source.

Value

the object specified by the <newobj> argument (or default name '<df.name>_subset'). 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.dataFrame() also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no

error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

DataSHIELD Development Team

`ds.densityGrid` *Generates a density grid with or without a priori defined limits*

Description

This function generates a grid density object which can then be used to produced heatmap or contour plots. The cells with a count > 0 and < nfilter.tab are considered invalid and the count is set to 0. The function prints the number of invalid cells in for participating study.

Usage

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

Arguments

`x`	a character the name of numerical vector
`y`	a character the name of numerical vector
`numints`	an integer, the number of intervals for the grid density object, by default is 20.
`type`	a character which represent the type of graph to display. If `type` is set to 'combine', a pooled grid density matrix is generated and one grid density matrix is generated for each study if `type` is set to 'split'.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

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 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).

Value

a grid density matrix is returned

Author(s)

Julia Isaeva, Amadou Gaye, Demetris Avraam for DataSHIELD Development Team

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")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Example1: generate a combined grid density object (the default behaviour)
  ds.densityGrid(x='D$LAB_TSC', y='D$LAB_HDL')

  # Example2: generate a grid density object for each study separately
  ds.densityGrid(x='D$LAB_TSC', y='D$LAB_HDL', type="split")

 # 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)

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


## End(Not run)
```

`ds.dim` *Retrieves the dimension of an object*

Description

this function is similar to R function `dim`

Usage

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

Arguments

`x`	a character, the name of R table object, for example a matrix, array or data frame
`type`	a character which 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
`checks`	a Boolean indicator of whether to undertake optional checks of model components. Defaults to checks=FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed

`datasources` a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

the function returns the unpooled or pooled dimension of the object by summing up the individual dimensions returned from each study or the dimension of the object in each study. Unlike the other DataSHIELD function the default behaviour is to output the dimension of each study separately.

Value

for an array, `NULL` or a vector of mode `integer`

Author(s)

Amadou Gaye, Julia Isaeva, Demetris Avraam, for DataSHIELD Development Team

See Also

ds.dataFrame to generate a table of type dataframe.

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:

  # load that contains the login details
  data(logindata)

  # login and assign all the stored variables.
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # Example 1: Get the dimension of the assigned datasets in each study
  ds.dim(x='D', type='combine')

  # Example 2: Get the pooled dimension of the assigned datasets
  ds.dim(x='D', type='combine')

  # Example 3: Get the dimension og the datasets in each single study
  # and the pooled dimension  - default
  ds.dim(x='D')

  # Example 4: Input has to be either matrix, data frame or an array
  # In the below example, the inpout is a vector so it will not work.
  ds.dim(x='D$LAB_TSC')

  # clear the Datashield R sessions and logout
```

```
  datashield.logout(opals)

## End(Not run)
```

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

Description

this function is similar to R function `exists`

Usage

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

Arguments

`x`	a character, the name of the object to look for.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

In DataSHIELD it is not possible to see the data sitting 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) son the server side. This function checks if an object do really exists on the server side. Further information about the object can be obtained using functions such as `ds.class`, `length` etc...

Value

a boolean, TRUE if the object is on the server side and FALSE otherwise

Author(s)

Gaye, A.

See Also

ds.class to check the type of an object.

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")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # assign 'LAB_TSC' in the dataframe D to a new variable 'labtsc'
  ds.assign(toAssign='D$LAB_TSC', newobj='labtsc')

  # now let us check if the variable 'labtsc' does now 'exist' on the server side
  ds.exists(x='labtsc')

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


## End(Not run)
```

---

ds.exp *Computes the exponential function*

Description

This function is similar to R function exp.

Usage

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

Arguments

x	a charcater, the name of a numerical vector.
newobj	the name of the new vector.If this argument is set to NULL, the name of the new variable is the name of the input variable with the suffixe '_exp' (e.g. 'PM_BMI_CONTINUOUS_exp', if input variable's name is 'PM_BMI_CONTINUOUS')
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

this is a wrapper that calls the exponential function on the server site.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Gaye, A.; Isaeva, J.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  myvar <- list("PM_BMI_CONTINUOUS")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # compute exponential function of the 'PM_BMI_CONTINUOUS' variable
  ds.exp(x='D$PM_BMI_CONTINUOUS')

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


## End(Not run)
```

---

ds.gee *Fits a Generalized Estimating Equation (GEE) model*

Description

A function that fits generalized estimated equations to deal with correlation structures arising from repeated measures on individuals, or from clustering as in family data.

Usage

```
ds.gee(formula = NULL, family = NULL, data = NULL,
  corStructure = "ar1", clusterID = NULL, startCoeff = NULL,
  userMatrix = NULL, maxit = 20, checks = TRUE, display = FALSE,
  datasources = NULL)
```

Arguments

formula	a string character, the formula which describes the model to be fitted.
family	a character, the description of the error distribution: 'binomial', 'gaussian', 'Gamma' or 'poisson'.

data	the name of the data frame that hold the variables in the regression formula.
corStructure	a character, the correlation structure: 'ar1', 'exchangeable', 'independence', 'fixed' or 'unstructure'.
clusterID	a character, the name of the column that hold the cluster IDs
startCoeff	a numeric vector, the starting values for the beta coefficients.
userMatrix	a list of user defined matrix (one for each study). These matrices are required if the correlation structure is set to 'fixed'.
maxit	an integer, the maximum number of iteration to use for convergence.
checks	a boolean, if TRUE (default) checks that takes 1-3min are carried out to verify that the variables in the model are defined (exist) on the server site and that they have the correct characteristics required to fit a GEE. If FALSE (not recommended if you are not an experienced user) no checks are carried except some very basic ones and eventual error messages might not give clear indications about the cause(s) of the error.
display	a boolean to display or not the intermediate results. Default is FALSE.
datasources	a list of opal object(s) obtained after login to opal servers; these objects also hold the data assigned to R, as a dataframe, from opal datasources.

Details

It enables a parallelized analysis of individual-level data sitting on distinct servers by sending commands to each data computer to fit a GEE model model. The estimates returned are then combined and updated coefficients estimate sent back for a new fit. This iterative process goes on until convergence is achieved. The input data should not contain missing values. The data must be in a data.frame obejct and the variables must be refer to through the data.frame.

Value

a list which contains the final coefficient estimates (beta values), the pooled alpha value and the pooled phi value.

Author(s)

Gaye, A.; Jones EM.

References

Jones EM, Sheehan NA, Gaye A, Laflamme P, Burton P. Combined analysis of correlated data when data cannot be pooled. Stat 2013; 2: 72-85.

See Also

ds.glm for genralized linear models

Examples

```
## Not run:

  # load the login data file for the correlated data
  data(geeLoginData)

  # login and assign all the stored variables to R
  opals <- datashield.login(logins=geeLoginData,assign=TRUE)

  # set some parameters for the function 9the rest are set to default values)
  myformula <- 'response~1+sex+age.60'
  myfamily <- 'binomial'
  startbetas <- c(-1,1,0)
  clusters <- 'id'
  mycorr <- 'ar1'

  # run a GEE analysis with the above specifed parameters
  ds.gee(data='D',formula=myformula,family=myfamily,corStructure=mycorr,clusterID=clusters,
         startCoeff=startbetas)

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


## End(Not run)
```

`ds.glm` *ds.glm calling glmDS1, glmDS2*

Description

Fits a generalized linear model (glm) on data from a single or multiple sources

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`	Denotes an object of class formula which is a character string describing the model to be fitted. 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 with a, b, c and d as covariates. By default all such models also include an intercept (regression constant) term. If all you need to fit are

straightforward models such as these, you do not need to read the remainder of this information about "formula". But if you need to fit a more complex model in a customised way, the following text gives a few additional pointers: As an example, the formula: "EVENT~1+TID+SEXF*AGE.60" denotes fit a glm with the variable "EVENT" as its outcome with covariates TID (in this case a 6 level factor [categorical] variable denoting "time period" with values between 1 and 6), SEXF (also a factor variable denoting sex and AGE.60 (a quantitative variable representing age-60 in years). The term "1" forces the model to include an intercept term which it would also have done by default (see above) but using "1" may usefully be contrasted with using "0" (as explained below), which removes the intercept term. The "*" between SEXF and AGE.60 means fit all possible main effects and interactions for and between those two covariates. As SEXF is a factor this is equivalent to writing SEXF+AGE.60+SEXF1:AGE.60 (the last element being the simple interaction term representing the product of SEXF level 1 [in this case female] and AGE.60). This takes the value 0 in all males (0 * AGE.60), and the same value as AGE.60 (1 * AGE.60) in females. If the formula had instead been written as: "EVENT~0+TID+SEXF*AGE.60" the 0 would mean do NOT fit an intercept term and, because TID happens to be a six level factor this would mean that the first six model parameters which were originally intercept+TID2+TID3+TID4+TID5+TID6 using the first formula will now become TID1+TID2+TID3+TID4+TID5+TID6. This is mathematically the same model, but conveniently, it means that the effect of each time period may now be estimated directly. For example, the effect of time period 3 is now obtained directly as the coefficient for TID3 rather than the sum of the coefficients for the intercept and TID3 which was the case using the original formula.

`data` A character string specifying the name of an (optional) dataframe that contains all of the variables in the glm formula. This avoids you having to specify the name of the dataframe in front of each covariate in the formula e.g. if the dataframe is called 'DataFrame' you avoid having to write: "DataFrame$y~DataFrame$a+DataFrame$b+ Processing stops if a non existing data frame is indicated.

`family` This argument identifies the error distribution function to use in the model. At present ds.glm has been written to fit family="gaussian" (i.e. a conventional linear model with normally distributed errors), family="binomial" (i.e. a conventional unconditional logistic regression model), and family = "poisson" (i.e. a Poisson regression model - of which perhaps the most commonly used application is for survival analysis using 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. For the majority of applications typically encountered in epidemiology and medical statistics, one these three classes of models will typically be what you need. However, if a particular user wishes us to implement an alternative family (e.g. 'gamma') or an alternative family/link combination (e.g. binomial with probit) we can discuss how best to meet that request: it will almost certainly be possible, but we may seek a small amount of funding or practical in-kind support from the user in order to ensure that it can be carried outin a timely manner

`offset` A character string specifying the name of a variable to be used as an offset. An offset is a component of a glm which may be viewed as a covariate with a known coefficient of 1.00 and so the coefficient does not need to be estimated by the model. As an example, an offset is needed to fit a piecewise exponential regression model. Unlike the standard glm() function in native R, ds.glm() only allows an offset to be set using the <offset> argument, it CANNOT be included directly in the formula via notation such as "y~a+b+c+d+offset(offset.vector.name)". So in ds.glm this model must be specified as: formula="y~a+b+c+d", ..., offset="offset.vector.name" and ds.glm then incorporates it appropriately into the formula itself.

`weights` A character string specifying the name of a variable containing prior regression weights for the fitting process. Like offset, ds.glm does not allow a weights vector to be written directly into the glm formula.

`checks` This argument is a boolean. If TRUE ds.glm then undertakes a series of checks of the structural integrity of the model that can take several minutes. Specifically it 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 a GLM. The default value is FALSE if an unexplained problem in the model fit is encountered.

`maxit` A numeric scalar denoting the maximum number of iterations that are permitted before ds.glm declares that the model has failed to converge. 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.

`CI` a numeric, the confidence interval.

`viewIter` a boolean, tells whether the results of the intermediate iterations should be printed on screen or not. Default is FALSE (i.e. only final results are shown).

`viewVarCov` a boolean indicating whether to return the variance-covariance matrix of parameter estimates. TRUE=yes, FALSE=no, default=FALSE

`viewCor` a boolean indicating whether to return the correlation matrix of parameter estimates. TRUE=yes, FALSE=no, default=FALSE

`datasources` specifies the particular opal object(s) to use, if it is not specified the default set of opals will be used. The default opals are always called default.opals. This parameter is set without inverted commas: e.g. datasources=opals.em or datasources=default.opals If you wish to specify the second opal server in a set of three, the parameter is specified: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set specify: e.g. datasources=opals.em[2,3]

Details

Fits a glm on data from a single source or from multiple sources. 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. e.g. 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.

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 EXACTLY the same ultimate sums

(4) CS then calculates sum(SCORES) 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.

Value

The main elements of the output returned by ds.glm are listed (below) as Example 1 under 'examples'.

Author(s)

Paul Burton for DataSHIELD Development Team

Examples

```
## Not run:
#  Example 1:
#The components of the standard output from ds.glm are listed below. Additional
#explanatory material for some
#components appears in square brackets in CAPITALIZED font. The model fitted here
#is a poisson regression model -
#a piecewise exponential regression model. It regresses a 1/0
#([died,failed]/[survived,censored]) numeric outcome held in the variable 'EVENT'
#on SEXF (a factor with male=0 and female=1) and AGE.60 ([age - 60] in years). The
#'*' in the formula denotes inclusion of both main effects
#and interactions for these covariates. When this formula is interpreted by ds.glm
#itself it separates these out for clarity
#when constructing the list of regression coefficients in the model. So, e.g., the
#final coefficient is named 'SEXF1:AGE.60' where
#the notation ':' indicates the interaction between SEXF and AGE.60. TID.f is a
#six level factor that allows the baseline hazard
#to vary across six different time periods that are precisely the same in each
#study: in this particular example the periods
#are 0-2 years, 2-3 years, 3-6 yrs, 6-6.5 years, 6.5-8 years and 8-10 years. The
#'1' in the formula explicitly forces the model
#to include an intercept, which means that the baseline (log) rate of death in
#each period may be obtained as the sum of
#the coefficient for the intercept + the coefficient for the corresponding period.
#So, for example, the baseline (log)rate
#of death in period 1 is approximately -0.988 + (-1.114) = -2.102. If this is
#exponentiated one obtains the value 0.1222
#an estimated event rate of 0.1222 events per annum. If the notation '0' had been
#used instead of '1', no intercept would
#have been fitted and the coefficient for period 2 would have been -2.102 directly
#estimating the baseline log rate in
#period 2.
#
#$Nvalid [TOTAL NUMBER OF VALID OBSERVATIONAL UNITS ACROSS ALL STUDIES]
#[1] 6254
#
#$Nmissing [TOTAL NUMBER OF OBSERVATIONAL UNITS ACROSS ALL STUDIES WITH AT LEAST
#ONE DATA ITEM MISSING]
#[1] 134
#
#$Ntotal [TOTAL OBSERVATIONAL UNITS ACROSS ALL STUDIES - SUM OF VALID AND MISSING UNITS]
#[1] 6388
#
#$disclosure.risk
#       RISK OF DISCLOSURE [THE VALUE 1 INDICATES THAT ONE OF THE DISCLOSURE TRAPS
#HAS BEEN TRIGGERED IN THAT STUDY]
#study1                  0
#study2                  0
```

```
#study3                  0
#
#$errorMessage
#       ERROR MESSAGES [EXPLANATION FOR ANY ERRORS OR DISCLOSURE RISKS IDENTIFIED]
#study1 "No errors"
#study2 "No errors"
#study3 "No errors"
#
#$nsubs [TOTAL NUMBER OF OBSERVATIONAL UNITS USED BY ds.glm - NB USUALLY SAME AS $Nvalid]
#[1] 6254
#
#$iter [TOTAL NUMBER OF ITERATIONS BEFORE CONVERGENCE ACHIEVED]
#[1] 9
#
#$family [ERROR FAMILY AND LINK FUNCTION]
#Family: poisson
#Link function: log
#
#
#$formula [MODEL FORMULA]
#[1] "EVENT ~ 1 + TID.f + SEXF * AGE.60 + offset(LOG.SURV)"
#
#
#[ESTIMATED COEFFICIENTS AND STANDARD ERRORS etc EXPANDED WITH ESTIMATED CONFIDENCE INTERVALS
#WITH % COVERAGE SPECIFIED BY CI= ARGUMENT. FOR POISSON MODEL, OUTPUT IS GENERATED
#ON SCALE OF LINEAR PREDICTOR (LOG RATES AND LOG RATE RATIOS)
#AND ON THE NATURAL SCALE AFTER EXPONENTIATION (RATES AND RATE RATIOS)]
#
#OUTPUT WRAPPED (AND MESSY) TO FIT IN WIDTH CONSTRAINT FOR R HELP FILE
#
#$coefficients
#               Estimate  Std. Error     z-value      p-value low0.95CI.LP high0.95CI.LP
#(Intercept)  -0.9883668593 0.040709331 -24.27863194 3.296958e-130 -1.068155681
#-0.908578037       0.37218402    0.34364172      0.4030970
#TID.f2       -1.1135769332 0.113156582  -9.84102663  7.494215e-23 -1.335359758
#-0.891794109       0.32838226    0.26306352      0.4099197
#TID.f3       -2.4132187489 0.145110041 -16.63026714  4.207143e-62 -2.697629203
#-2.128808294       0.08952667    0.06736503      0.1189790
#TID.f4       -0.9834247291 0.202670688  -4.85232836  1.220204e-06 -1.380651979
#-0.586197480       0.37402796    0.25141458      0.5564391
#TID.f5       -1.2752840562 0.152704468  -8.35132119  6.750358e-17 -1.574579313
#-0.975988799       0.27935161    0.20709466      0.3768196
#TID.f6       -1.0459413608 0.141295558  -7.40250701  1.336371e-13 -1.322875566
#-0.769007156       0.35136091    0.26636824      0.4634730
#SEXF1        -0.6239830856 0.060097042 -10.38292508  2.965473e-25 -0.741771124
#-0.506195048       0.53580602    0.47626964      0.6027848
#AGE.60        0.0428295263 0.002671504  16.03198917  7.639759e-58  0.037593474
#0.048065578       1.04375995    1.03830905      1.0492395
#SEXF1:AGE.60 -0.0003408707 0.004111513  -0.08290639  9.339260e-01 -0.008399288
#0.007717547       0.99965919    0.99163589      1.0077474
#
#$dev [RESIDUAL DEVIANCE]
#[1] 5266.551
```

```
#
#$df [RESIDUAL DEGREES OF FREEDOM - NB RESIDUAL DEGREES OF FREEDOM + NUMBER OF
#PARAMETERS IN MODEL = $nsubs]
#[1] 6245
#
#$output.information [REMINDER TO USER THAT THERE IS MORE INFORMATION AT THE TOP OF THE OUTPUT]
#[1] "SEE TOP OF OUTPUT FOR INFORMATION ON MISSING DATA AND ERROR MESSAGES"
#
#  Example 2:
#EXAMPLE 2 DIFFERS FROM EXAMPLE 1 PRIMARILY IN HAVING '0' RATHER THAN '1' IN THE
#MODEL FORMULA. THIS FORCES THE MODEL
#NOT TO INCLUDE A REGRESSION INTERCEPT. BECAUSE THE SECOND COVARIATE TID.f IS A
#FACTOR, REMOVAL OF THE INTERCEPT IN
#THIS WAY MEANS THAT THE REGRESSION PARAMETERS ASSOCIATED WITH THE TID.F COVARIATE
#EACH DIRECTLY ESTIMATE THE LOG RATE IN
#EACH TIME PERIOD. SO, FOR EXAMPLE, TID.f2 DIRECTLY ESTIMATES THE LOG RATE IN
#PERIOD 2, I.E. 2.1019 WHICH IS PRECISELY
#EQUIVALENT TO THE ESTIMATE DERIVED FROM THE SUM OF INTERCEPT AND TID.f2 IN
#EXAMPLE 1 (ABOVE).
#
#
#$nsubs
#[1] 6254
#
#$iter
#[1] 9
#
#$family
#Family: poisson
#Link function: log
#
#
#$formula
#[1] "EVENT ~ 0 + TID.f + SEXF * AGE.60 + offset(LOG.SURV)"
#
#
#OUTPUT WRAPPED (AND MESSY) TO FIT IN WIDTH CONSTRAINT FOR R HELP FILE
#
#$coefficients
#                  Estimate  Std. Error      z-value       p-value low0.95CI.LP
#high0.95CI.LP EXPONENTIATED RR low0.95CI.EXP high0.95CI.EXP
#TID.f1       -0.9883668593 0.040709331 -24.27863194 3.296958e-130 -1.068155681
#-0.908578037       0.37218402    0.34364172     0.40309701
#TID.f2       -2.1019437924 0.111730815 -18.81257014  5.957806e-79 -2.320932166
#-1.882955419       0.12221863    0.09818202     0.15213980
#TID.f3       -3.4015856082 0.144060220 -23.61224772 2.884936e-123 -3.683938452
#-3.119232765       0.03332039    0.02512383     0.04419106
#TID.f4       -1.9717915884 0.201948314  -9.76384278  1.609388e-22 -2.367603011
#-1.575980166       0.13920723    0.09370507     0.20680475
#TID.f5       -2.2636509154 0.151818841 -14.91021073  2.828585e-50 -2.561210376
#-1.966091454       0.10397020    0.07721123     0.14000300
#TID.f6       -2.0343082201 0.140582544 -14.47056064  1.859503e-47 -2.309844942
#-1.758771498       0.13077092    0.09927664     0.17225635
```

```
#SEXF1        -0.6239830856 0.060097042 -10.38292508  2.965473e-25 -0.741771124
#-0.506195048       0.53580602    0.47626964     0.60278479
#AGE.60        0.0428295263 0.002671504  16.03198917  7.639759e-58  0.037593474
#0.048065578       1.04375995    1.03830905     1.04923946
#SEXF1:AGE.60 -0.0003408707 0.004111513  -0.08290639  9.339260e-01 -0.008399288
#0.007717547       0.99965919    0.99163589     1.00774740
#
#$dev
#[1] 5266.551
#
#$df
#[1] 6245
#
#$output.information
#[1] "SEE TOP OF OUTPUT FOR INFORMATION ON MISSING DATA AND ERROR MESSAGES"
#
#  Example 3:
#EXAMPLE 3 DIFFERS FROM EXAMPLE 2 PRIMARILY BECAUSE IT INCLUDES A REGRESSION
#WEIGHT VECTOR. IN THIS PARTICULAR
#CASE THE VECTOR WEIGHTS4 IS SIMPLY A VECTOR OF 4s AND, FROM A THEORETICAL
#PERSPECTIVE, THIS SHOULD SIMPLY MAKE EACH SINGLE
#OBSERVATION EQUIVALENT TO 4 OBSERVATIONS. THIS SHOULD INCREASE THE DEVIANCE AND
#RESIDUAL DEVIANCE BY A FACTOR OF FOUR
#WHICH IT DOES (5266.551*4=21066.2). FURTHERMORE, BECAUSE THE STANDARD ERROR OF
#PARAMETER ESTIMATES IS INVERSELY RELATED TO THE
#SQUARE ROOT OF THE NUMBER OF OBSERVATIONS, EACH STANDARD ERROR SHOULD BE HALVED
#(1/sqrt(4) = 0.5) WHICH CAN AGAIN BE SEEN
#TO BE TRUE, AND THE ESTIMATED VALUE OF z-STATISTICS DOUBLED.
#
#$iter

#[1] 9
#
#$family
#
#Family: poisson
#Link function: log
#
#
#$formula
#[1] "EVENT ~ 0 + TID.f + SEXF * AGE.60 + offset(LOG.SURV) + weights(WEIGHTS4)"
#
#
#OUTPUT WRAPPED (AND MESSY) TO FIT IN WIDTH CONSTRAINT FOR R HELP FILE
#
#$coefficients
#                  Estimate  Std. Error     z-value       p-value low0.95CI.LP
#high0.95CI.LP EXPONENTIATED RR low0.95CI.EXP high0.95CI.EXP
#TID.f1       -0.9883668593 0.020354665 -48.5572639  0.000000e+00  -1.02826127
#-0.948472448       0.37218402    0.35762824     0.38733224
#TID.f2       -2.1019437924 0.055865407 -37.6251403  0.000000e+00  -2.21143798
#-1.992449606       0.12221863    0.10954301     0.13636099
#TID.f3       -3.4015856082 0.072030110 -47.2244954  0.000000e+00  -3.54276203
```

```
#-3.260409187       0.03332039    0.02893330     0.03837269
#TID.f4       -1.9717915884 0.100974157 -19.5276856  6.386915e-85  -2.16969730
#-1.773885877       0.13920723    0.11421218     0.16967238
#TID.f5       -2.2636509154 0.075909421 -29.8204215 2.123826e-195  -2.41243065
#-2.114871185       0.10397020    0.08959725     0.12064883
#TID.f6       -2.0343082201 0.070291272 -28.9411213 3.629743e-184  -2.17207658
#-1.896539859       0.13077092    0.11394076     0.15008704
#SEXF1        -0.6239830856 0.030048521 -20.7658502  8.815359e-96  -0.68287710
#-0.565089067       0.53580602    0.50516150     0.56830953
#AGE.60        0.0428295263 0.001335752  32.0639783 1.401856e-225   0.04021150
#0.045447552       1.04375995    1.04103093     1.04649612
#SEXF1:AGE.60 -0.0003408707 0.002055757  -0.1658128  8.683043e-01  -0.00437008
#0.003688338       0.99965919    0.99563946     1.00369515
#
#$dev
#[1] 21066.2
#
#$df
#[1] 6245
#
#$output.information
#[1] "SEE TOP OF OUTPUT FOR INFORMATION ON MISSING DATA AND ERROR MESSAGES"
#
# load the file that contains the login details
#data(glmLoginData)
#
# login and assign all the variables to R
#opals <- datashield.login(logins=glmLoginData, assign=TRUE)
#
# EXAMPLE 4: RUN A LOGISTIC REGRESSION WITHOUT INTERACTION (E.G. DIABETES
#PREDICTION USING BMI AND HDL LEVELS AND GENDER)
#mod <- ds.glm(formula='D$DIS_DIAB~D$GENDER+D$PM_BMI_CONTINUOUS+D$LAB_HDL',
#family='binomial')
#mod
#
# EXAMPLE 5: RUN THE SAME GLM MODEL WITHOUT AN INTERCEPT
# (produces separate baseline estimates for Male and Female)
#mod <- ds.glm(formula='D$DIS_DIAB~0+D$GENDER+D$PM_BMI_CONTINUOUS+D$LAB_HDL',
#family='binomial')
#mod
#
# EXAMPLE 6: RUN THE SAME GLM MODEL WITH AN INTERACTION BETWEEN GENDER AND
#PM_BMI_CONTINUOUS
#mod <- ds.glm(formula='D$DIS_DIAB~D$GENDER*D$PM_BMI_CONTINUOUS+D$LAB_HDL',
#family='binomial')
#mod
#
# Example 7: FIT A STANDARD GAUSSIAN LINEAR MODEL WITH AN INTERACTION
#mod <- ds.glm(formula='D$PM_BMI_CONTINUOUS~D$DIS_DIAB*D$GENDER+D$LAB_HDL',
#family='gaussian')
#mod
#
# clear the Datashield R sessions and logout
```

```
#datashield.logout(opals)

## End(Not run)
```

---

ds.glmSLMA *ds.glmSLMA calling glmSLMADS1, glmSLMADS2*

Description

Fits a generalized linear model (glm) on data from a single or multiple sources with pooled co-analysis across studies being based on study level meta-analysis

Usage

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

Arguments

formula	Denotes an object of class formula which is a character string describing the model to be fitted. 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 with a, b, c and d as covariates. By default all such models also include an intercept (regression constant) term. If all you need to fit are straightforward models such as these, you do not need to read the remainder of this information about "formula". But if you need to fit a more complex model in a customised way, the following text gives a few additional pointers: As an example, the formula: "EVENT~1+TID+SEXF*AGE.60" denotes fit a glm with the variable "EVENT" as its outcome with covariates TID (in this case a 6 level factor [categorical] variable denoting "time period" with values between 1 and 6), SEXF (also a factor variable denoting sex and AGE.60 (a quantitative variable representing age-60 in years). The term "1" forces the model to include an intercept term which it would also have done by default (see above) but using "1" may usefully be contrasted with using "0" (as explained below), which removes the intercept term. The "*" between SEXF and AGE.60 means fit all possible main effects and interactions for and between those two covariates. As SEXF is a factor this is equivalent to writing SEXF+AGE.60+SEXF1:AGE.60 (the last element being the simple interaction term representing the product of SEXF level 1 [in this case female] and AGE.60). This takes the value 0 in all males (0 * AGE.60), and the same value as AGE.60 (1 * AGE.60) in females. If the formula had instead been written as: "EVENT~0+TID+SEXF*AGE.60" the 0 would mean do NOT fit an intercept term and, because TID happens to be a six level factor this would mean that the first six model parameters which were originally intercept+TID2+TID3+TID4+TID5+TID6

using the first formula will now become TID1+TID2+TID3+TID4+TID5+TID6. This is mathematically the same model, but conveniently, it means that the effect of each time period may now be estimated directly. For example, the effect of time period 3 is now obtained directly as the coefficient for TID3 rather than the sum of the coefficients for the intercept and TID3 which was the case using the original formula.

`family` This argument identifies the error distribution function to use in the model. At present ds.glm has been written to fit family="gaussian" (i.e. a conventional linear model with normally distributed errors), family="binomial" (i.e. a conventional unconditional logistic regression model), and family = "poisson" (i.e. a Poisson regression model - of which perhaps the most commonly used application is for survival analysis using 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. For the majority of applications typically encountered in epidemiology and medical statistics, one these three classes of models will typically be what you need. However, if a particular user wishes us to implement an alternative family (e.g. 'gamma') or an alternative family/link combination (e.g. binomial with probit) we can discuss how best to meet that request: it will almost certainly be possible, but we may seek a small amount of funding or practical in-kind support from the user in order to ensure that it can be carried outin a timely manner

`offset` A character string specifying the name of a variable to be used as an offset. An offset is a component of a glm which may be viewed as a covariate with a known coefficient of 1.00 and so the coefficient does not need to be estimated by the model. As an example, an offset is needed to fit a piecewise exponential regression model. Unlike the standard glm() function in native R, ds.glmSLMA() only allows an offset to be set using the <offset> argument, it CANNOT be included directly in the formula via notation such as "y~a+b+c+d+offset(offset.vector.name)". So in ds.glmSLMA this model must be specified as: formula="y~a+b+c+d", ..., offset="offset.vector.name" and ds.glmSLMA then incorporates it appropriately into the formula itself.

`weights` A character string specifying the name of a variable containing prior regression weights for the fitting process. Like offset, ds.glmSLMA does not allow a weights vector to be written directly into the glm formula.

`combine.with.metafor`
This argument is Boolean. If TRUE (the default) 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).

`dataName` A character string specifying the name of an (optional) dataframe that contains all of the variables in the glm formula. This avoids you having to specify the name of the dataframe in front of each covariate in the formula e.g. if the dataframe is called "DataFrame" you avoid having to write: "DataFrame$y~DataFrame$a+DataFrame$b+ Processing stops if a non existing data frame is indicated.

`checks` This argument is a boolean. If TRUE ds.glmSLMA then undertakes a series

of checks of the structural integrity of the model that can take several minutes. Specifically it 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 a GLM. The default value is FALSE and so it is suggested that the argument <checks> is only made TRUE if an unexplained problem in the model fit is encountered.

`maxit` A numeric scalar denoting the maximum number of iterations that are permitted before ds.glm declares that the model has failed to converge. 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 choose to set maxit=30 - but depending on the nature of the models you wish to fit, you may wish to be alerted more quickly than this if there is a delay in convergence, or you may wish to allow MORE iterations.

`datasources` specifies the particular opal object(s) to use, if it is not specified the default set of opals will be used. The default opals are always called default.opals. This parameter is set without inverted commas: e.g. datasources=opals.em or datasources=default.opals If you wish to specify the second opal server in a set of three, the parameter is specified: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set specify: e.g. datasources=opals.em[c(2,3)]

Details

ds.glmSLMA specifies the structure of a generalized linear model (glm) to be fitted separately on each study/data source. The model is first constructed and disclosure checked by glmSLMADS1. This aggregate function then returns its output to ds.glmSLMA which processes the information and uses it in a call to the second aggregate function glmSLMADS2. This call specifies and fits the required glm in each data source. Unlike glmDS2 (called by the more commonly used generalized linear modelling client-side function ds.glm) the requested model is then fitted to completion on the data in each study rather than iteration by iteration on all studies combined. At the end of this SLMA fitting process glmSLMADS2 returns study-specific parameter estimates and standard errors to the client. These can then be pooled using random effects (or fixed effects) meta-analysis - eg using the metafor package. This mode of model fitting may reasonably be called study level meta-analysis (SLMA) although the analysis is based on estimates and standard errors derived from direct analysis of the individual level data in each study rather than from published study summaries (as is often the case with SLMA of clinical trials etc). Furthermore, unlike common approaches to study-level meta-analysis adopted by large multi-study research consortia (eg in the combined analysis of identical genomic markers across multiple studies), the parallel analyses (in every study) under ds.glmSLMA are controlled entirely from one client. This avoids the time-consuming need to ask each study to run its own analyses and the consequent necessity to request additional work from individual studies if the modelling is to be extended to include analyses not subsumed in the original analytic plan. Additional analyses of this nature may, for example, include analyses based on interactions between covariates identified as having significant main effects in the original analysis. From a mathematical perspective, the SLMA approach (using ds.glmSLMA) differs fundamentally from the usual approach using ds.glm in that the latter is 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 R. However, although this may

sound to be preferable under all circumstances, the SLMA approach actually offers key inferential advantages when there is marked heterogeneity between sources that cannot simply be corrected with fixed effects each reflecting a study or centre-effect. In particular, fixed effects cannot simply be used in this way when there there is heterogeneity in the effect that is of scientific interest.

Value

Many of the elements of the output list returned by ds.glmSLMA from each study separately are precisely 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. The return results from each separate study appear first in the return list with the full set of results from each study presented in a block and the blocks listed in the order in which the studies appear in <datasources>. As regards the elements within each study the most important elements are included last in the return list because they then appear at the bottom of a simple print out of the return object. In reverse order, these key elements are listed below. In addition to the elements reflecting the primary results of the analysis, ds.glmSLMA also returns a range of error messages if the model fails indicating why failure may have occurred and in particular detailing any disclosure traps that may have been

coefficients:- a matrix in which the first column contains the names of all of the regression parameters (coefficients) in the model, the second column contains the estimated values of the coefficients (called estimates), the third the corresponding standard errors, the fourth the ratio corresponding to the value of each estimate divided by its standard error and the fifth the p-value treating that ratio as a standardised normal deviate (a simple Wald test).

family:- indicates the error distribution and link function used in the glm

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 vector (if any) holding regression weights

offset:- 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 NAs. Usually, na.action=na.omit indicating any individual (or more strictly any "observational unit") that has any data missing that are needed for the model is exluded from the fit, even if all the rest of the required data are present. These required data include: the outcome variable, covariates, or any values in a regression weight vector or offset vector. As a side effect of this, when you include additional covariates in model you may exclude extra individuals from the analysis and this can seriously distort inferential tests based on assuming models are nested (eg likelihood ratio tests).

iter:- the number of iterations required to achieve convergence file for the glm() function in native R.

input.beta.matrix.for.SLMA:- a matrix containing the vector of coefficient estimates from each study. In combination with the corresponding standard errors (see input.se.matrix.for.SLMA) these can be imported directly into a study level meta-analysis (SLMA) package such as metafor to generate estimates pooled via SLMA

input.se.matrix.for.SLMA:- a matrix containing the vector of standard error estimates for coefficients from each study. In combination with the coefficients (see input.beta.matrix.for.SLMA) these can be imported directly into a study level meta-analysis (SLMA) package such as metafor to generate estimates pooled via SLMA

SLMA.pooled.estimates:- if the argument <combine.with.metafor> = TRUE, ds.glmSLMA also returns 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)

there are a small number of more esoteric items of information returned by ds.glmSLMA. Additional information about these can be found in the help

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.heatmapPlot` *Generates a heatmap plot*

Description

Generates a heatmap 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, the name of a numerical vector
y	a character, the name of a numerical vector
type	a character which represents the type of graph to display. If type is set to 'combine', a combined heatmap plot displayed and if type is set to 'split', each heatmap is plotted separately.
show	a character which 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	a number of intervals for a density grid object.
method	a character which defines which heatmap will be created. If method is set to 'smallCellsRule' (default option), the heatmap 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 heatmap of the scaled centroids of each k nearest neighbours 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 heatmap 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.
k	the number of the nearest neghbours for which their centroid is calculated. 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'.
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'. 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.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

The 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 minumum and maximum values but rather close approximates of the real minimum and maximum value. This was done to reduce the risk of potential disclosure.

Value

a heatmap plot

Author(s)

Julia Isaeva, Amadou Gaye, Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # load the file that contains the login details
  data(logindata)

  # login and assign the required variables to R
  opals <- datashield.login(logins=logindata, assign=TRUE)

 # Example 1: Plot a combined (default behaviour) heatmap plot of the variables 'LAB_TSC'
 # and 'LAB_HDL' using the method 'smallCellsRule' (default method) that applies a stochastic
  # noise in the extreme values of the variables' range.
 ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL')

 # Example 2: the same as example 1
 ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="smallCellsRule", type='combine')

 # Example 3: similar as example 2 but for type='split'
 ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="smallCellsRule", type='split')

 # Example 4: Plot a combined (default behaviour) heatmap plot of the variables 'LAB_TSC'
 # and 'LAB_HDL' using the method 'deterministic' that plots the exact heatmap plot of the
  # centroids of each 3 (default number) nearest neighbours.
 ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic")

 # Example 5: the same as example 4
ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic", k=3, type='combine')

 # Example 6: similar as example 5 for type='split'
ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic", k=3, type='split')

 # Example 7: similar as example 6 for k=7
ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="deterministic", k=7, type='split')

 # Example 8: similar as example 7 for numints=40
 ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', numints=40, method="deterministic", k=7,
                type='split')

 # Example 9: Plot a combined (default behaviour) heatmap plot of the variables 'LAB_TSC'
 # and 'LAB_HDL' using the method 'probabilistic' that plots the exact heatmap plot of the
  # noisy data
 ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic")

 # Example 10: the same as example 9
```

```
ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic", noise=0.25, type='combine')

# Example 11: the same as example 10 but for bigger level of noise
ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic", noise=2, type='combine')

# Example 12: the same as example 11 but for type='split'
ds.heatmapPlot(x='D$LAB_TSC', y='D$LAB_HDL', method="probabilistic", noise=2, type='split')

# Example 13: if any of the input variables is a factor then the function fails
ds.heatmapPlot(x='D$LAB_TSC', y='D$GENDER')

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


## End(Not run)
```

`ds.histogram` *Generates a histogram plot*

Description

This function plots a non-disclosive histogram

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 charcater, the name of the vector of values for which the histogram is desired.
`type`	a character which represent the type of graph to display. If `type` is set to 'combine', a histogram that merges the single plot is displayed. Each histogram is plotted separately if `type` is set to 'split'.
`num.breaks`	a numeric specifying the number of breaks of the histogram. The default value is set to 10.
`method`	a character which defines which histogram will be created. If `method` is set to 'smallCellsRule' (default option), the histogram of the actual variable is created but bins with low counts are removed. If `method` is set to '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. If the `method` is set to 'probabilistic', then 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`.

k — the number of the nearest neghbours for which their centroid is calculated. 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'.

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'. 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.

vertical.axis, — a character which defines what is shown in the vertical axis of the plot. If `vertical.axis` is set to 'Frequency' then the histogram of the frequencies is returned. If `vertical.axis` is set to 'Density' then the histogram of the densities is returned.

datasources — a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

It calls a datashield server side function that produces the histogram objects to plot. Two options are possible as identified by the argument `method`. The first option creates a histogram that excludes bins with counts smaller than the allowed threshold. The second option creates a histogram of the centroids of each k nearest neighbours. The function allows for the user to plot disctinct histograms (one for each study) or a combine histogram that merges the single plots.

Value

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

Author(s)

Amadou Gaye, Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login to the servers
  opals <- opal::datashield.login(logins=logindata, assign=TRUE)

  # Example 1: generate a histogram for each study separately (the default behaviour)
```

```
ds.histogram(x='LD$PM_BMI_CONTINUOUS', type="split")

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

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

# Example 4: generate a combined histogram with the deterministic method
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='deterministic', type='combine')

# Example 5: same as Example 4 but with k=50
ds.histogram(x='LD$PM_BMI_CONTINUOUS', k=50, method='deterministic', type='combine')

# Example 6: same as Example 4 but with k=1740 (here we see that as k increases we have
             big utility loss)
ds.histogram(x='LD$PM_BMI_CONTINUOUS', k=1740, method='deterministic', type='combine')

# Example 7: same as Example 6 but for split analysis
ds.histogram(x='LD$PM_BMI_CONTINUOUS', k=1740, method='deterministic', type='split')

# Example 7: if k is less than the pre-specified threshold then the function returns an error
ds.histogram(x='LD$PM_BMI_CONTINUOUS', k=2, method='deterministic')

# Example 8: generate a combined histogram with the probabilistic method
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', type='combine')

# Example 9: generate a histogram with the probabilistic method for each study separately
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', type='split')

# Example 10: same as Example 9 but with higher level of noise
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', noise=0.5, type='split')

# Example 11: if 'noise' is less than the pre-specified threshold then the function returns
              an error
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', noise=0.1, type='split')

# Example 12: same as Example 9 but with bigger number of breaks
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', type='split', num.breaks=30)

# Example 13: same as Example 12 but the vertical axis shows densities instead of frequencies
ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', type='split', num.breaks=30,
               vertical.axis='Density')

# Example 14: create a histogram and the probability density on the plot
hist <- ds.histogram(x='LD$PM_BMI_CONTINUOUS', method='probabilistic', type='combine',
                     num.breaks=30, vertical.axis='Density')
lines(hist$mids, hist$density)

# clear the Datashield R sessions and logout
opal::datashield.logout(opals)
```

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

---

ds.isNA *Checks if a 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, the name of the vector to check.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources. By default an internal function looks for 'opal' objects in the environment and sets this parameter.

Details

In certain analyses such as GLM none of the variable 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.

Value

a boolean 'TRUE' if the vector is empty (all values are 'NA') and 'FALSE' otherwise.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load the login data
  data(logindata)

  # login and assign specific variable(s)
  myvar <- list("LAB_HDL")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)
```

```
  # check if all the observation of the variable 'LAB_HDL' are missing (NA)
  ds.isNA(x='D$LAB_HDL')

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


## End(Not run)
```

`ds.isValid` *Checks if an 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, the name of a vector, dataframe or matrix.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources. By default an internal function looks for 'opal' objects in the environment and sets this parameter.

Details

In DataSHIELD, analyses are possible only on valid objects to ensure the ouput is not disclosive. This function checks if an input object is valid. A vector is valid if the number of observations are 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 dataframe or a matrix is valid if the number of rows is equal or greater than the set threshold.

Value

a boolean, TRUE if input object is valid and FALSE otherwise.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  myvar <- list("LAB_TSC", "GENDER")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Example 1: Check if the dataframe assigned above is valid
  ds.isValid(x='D')

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


## End(Not run)
```

---

ds.length *Gets the length of a vector or list*

Description

This function is similar to R function `length`.

Usage

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

Arguments

`x`	a string character, the name of a vector or list
`type`	a character which 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
`checks`	a Boolean indicator of whether to undertake optional checks of model components. Defaults to checks=FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

The function returns the pooled length of the a vector or a list, or the length of the a vector or a list for each study separately.

Value

a numeric, the length of the input vector or list.

Author(s)

Amadou Gaye, Demetris Avraam, for DataSHIELD Development Team

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the variables stored on the server side
  opals <- datashield.login(logins=logindata, assign=TRUE)

  # 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')

  # 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')

  # 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')

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


## End(Not run)
```

---

ds.levels *Returns the levels attribute of a factor*

Description

This function is similar to R function `levels`

Usage

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

Arguments

x	a character, the name of a factor variable
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

This is just a wrapper function for the server side function.

Value

levels of x

Author(s)

Gaye, A.; Isaeva, J.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # Example 1: Get the levels of the PM_BMI_CATEGORICAL variable
  ds.levels(x='D$PM_BMI_CATEGORICAL')

 # Example 2: Get the levels of the LAB_TSC  SHOULD NOT WORK AS IT IS A CONTINUOUS VARIABLE
  ds.levels(x='D$LAB_TSC')

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


## End(Not run)
```

---

ds.lexis *ds.lexis calling lexisDS1, lexisDS2, lexisDS3*

Description

Takes a dataframe 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. For each survival interval the expanded dataframe contains variables denoting the surival time and the censoring status in that specific interval, a unique ID for every time interval and carries copies of other IDs and variables. 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 unecessary 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 time periods is a given study is less than the specified disclosure filter determining minimum acceptable cell size in a table (nfilter.tab) then the expanded dataframe is not created in that study, and a studyside message to this effect is made available in that study via ds.message()

Usage

```
ds.lexis(data = NULL, intervalWidth = NULL, idCol = NULL,
  entryCol = NULL, exitCol = NULL, statusCol = NULL,
  variables = NULL, expandDF = NULL, datasources = NULL)
```

Arguments

`data`	is a character string. This specifies the name of a dataframe containing the survival data to be expanded. Often, the dataframe will also hold the original total-survival-time and final-censoring variables but the lexis function is deliberately set up so those can also be specified as coming either from a different data frame or from the root area of your analysis (i.e. lying outside any dataframe)table that holds the original data, this is the data to be expanded.
`intervalWidth`	is a numeric vector specifying the length of each interval. 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 a number of disclosure checks (see details).
`idCol`	is a character string denoting the name of the column that holds the individual IDs of the subjects. This may be numeric or character. Note that when a particular variable is identified as being the main ID to Opal when the data are first transferred to Opal (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. So, in a dataset containing sequential IDs 1-1000, the order allocated by Opal will be mean that the first

thirteen rows in the original dataset (before expansion) will correspond to the original IDs: 1,10,100,101,102,103,104,105,106,107,108,109,11 (analogous to b, ba, baa, bab, bac, bad, bae, baf, bag, bah, bai, bb ...) 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 own unique numeric vector that is allocated from 1:M (where there are M individuals) in whatever order they appear in the original dataset. It is this ds.lexis sequentially allocated numeric ID vector that is ultimately combined with the interval period number to produce the expanded unique ID variable in the expanded data set (e.g. see 77.13 under 'details')

`entryCol` is a character string denoting the name of the column that holds the entry times (i.e. start of follow up). 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 (as is most common) 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.

`exitCol` is a character string denoting the name of the column that holds the exit times (i.e. end of follow up). If the entry times are set, or defaulted, to zero, the exitCol variable should contain the total survival times.

`statusCol` is a character string denoting the name of the column that holds the failure/censoring status of each subject (see under details).

`variables` is a vector of character strings denoting the column names of additional variables to include in the final expanded table. If the 'variables' argument is not set (is null) but the 'data' argument is set the expanded data set will contain all variables in the dataframe 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.

`expandDF` is a character string denoting the name of the new dataframe containing the expanded data set. If you specify a name, that name will be used, but if no name is specified it will be defaulted to 'name of dataframe specified by data argument' with '_expanded' as a suffix. If you use the client side function ds.ls() after running ds.lexis the new dataframe you have created should be listed, and the output of the function advises you to do this. If the function call fails (e.g. the expanded dataframe does not appear when you run ds.ls() you can use the command ds.message("messageobj") and depending what has gone wrong, there may be an explanatory error message that ds.message("messageobj") will reveal. Errors arising directly from deliberate disclosure traps are explained under details.

`datasources` requires specification of one or more opal objects. As in all client-side functions,

a list of opal object(s) obtained after login to opal servers; these objects also hold the data assigned to R, as a `data frame`, from opal datasources

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 how short or long that period may be. Each row includes: (1) a variable (CENSOR) indicating failure status for a particular interval in that interval also known as censoring status (1=failed, died, relapsed, developed a disease etc, 0= e.g. lost-to-follow-up or passed right through the interval without failing); (2) an exposure-time variable (SURVTIME) 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) The expanded data set also includes a unique ID (UID.expanded)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 (in whatever order they have been placed by Opal [which is often different to the original numeric order of the IDs that were actually specified to Opal]) 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) In addition to UID.expanded, the expanded dataframe also includes a simpler variable IDSEQ which is simply 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) Finally, the expanded dataset contains any other variables pertaining to each individual that the user would like to carry forwarded to a survival analysis based on the expanded data. Typically, this will include the original ID as specified to Opal, 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. The clientside function ds.lexis calls three server side functions. First lexisDS1 which is an aggregate function. This identifies the maximum survival time in each study (with a positive random value added to prevent disclosure). When these are all returned to the clientside, the maximum of these maxima us selected and this ensures that the end of the final exposure period will always include all of the events in all studies. The value of this maximum maximum is returned as part of the output of ds.lexis - REMEMBER IT INCLUDES A RANDOM ADDITION SO IT WILL ALWAYS BE LARGER THAN THE ACTUAL LARGEST SURVIVAL TIME BUT THIS DOESN'T MATTER TOO MUCH THOUGH IF IT IS LARGE RELATIVE TO THE REAL LENGTH OF THE FINAL SURVIVAL PERIOD, IT WILL DISTORT (REDUCE) THE ESTIMATE OF THE BASELINE HAZARD IN THE FINAL TIME PERIOD. IN THE UNLIKELY EVENT THAT THIS IS A PROBLEM FOR ANYONE, WE COULD EXPLORE A WORK ROUND IN A LATER VERSION OF DataSHIELD. Second, lexisDS2 undertakes the actual expansion to produce the new dataframe. The function lexisDS2, which is an aggregate function, also checks the arguments to identify disclosure risks. This includes any attempts to send illegal character strings to the serverside as part of the intervalWidth argument. It also checks that the total length of the intervalWidth vector (effectively the total number of intervals

specified) does not exceed nfilter.glm*length of vectors in the collapsed dataframe (before expansion to produce expandDF. This is because intervalWidth defines a numeric vector (completely determined by the user) which might be used to create and define subsets if the vector you defined was the same length as the primary data vectors in the model. In addition lexisDS2 checks the number of failures in each time interval and if one or more intervals in a study contain fewer than the value of nfilter.tab (the minimum valid non-zero cell count in a table) set by the server administrator for that study, the test will be failed for the relevant server. If any of these tests are failed, creation of the expanded dataframe will be blocked and an explanatory error message will be stored on each server. These messages can then be read using the command: ds.message("messageobj"). Third, the assign function lexisDS3 simplifies the final output so that the object specified by the expandDF argument is the actual dataframe rather than a table within a list.

Value

The function returns a dataframe which is the expanded version of the input table. The required dataframe is created on each of the study servers not on the client - which is why you need to use ds.ls() to see it. If a expandDF argument was specified, this defines the name of the expanded dataframe. For example, expandDF="charlie" will create an expanded dataframe called charlie on each study server. If the expandDF argument is not set, the expanded dataframe is, by default, named by combining the name of the original collapsed dataframe (as specified by the data argument) with '_expanded'. So if data = "alice" and expandDF is not set, the expanded dataframe will be called alice_expanded on each study server.

Author(s)

Burton PR, Gaye A

See Also

ds.glm for genralized linear models

ds.gee for generalized estimating equation models

Examples

```
## Not run:
#EXAMPLE 1
#In this example, the data to be expanded are held in a dataframe called 'CM'. 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. The original ID, entry-time, exit-time and censoring
#variables are all included in the original dataframe in variables CM$ID, CM$STARTTIME,
#CM$ENDTIME and CM$CENS. The expanded dataframe will be created with the name "EM$new" and the
#data will be held in a dataframe called 'expanded.table' inside EM$new.
#
#ds.lexis.5(data = "CM", intervalWidth = 2.5, idCol = "CM$ID", entryCol = "CM$STARTTIME",
#          exitCol = "CM$ENDTIME", statusCol = "CM$CENS", expandDF = "EM.new")
#ds.ls() #to confirm expanded dataframe created
#
#Please note, if the censoring variable had instead been held in a variable 'CENSOR' (outside
#any dataframe) or in DF$died (inside a different dataframe called DF) then it would have been
#perfectly acceptable to specify statusCol="CENSOR" and/or statusCol="DF$died"
```

```
#
#For illustration, the following is a schema of a typical set of variables you get in an
#expanded dataframe. Depending how the arguments of ds.lexis are specified some variables may be
#repeated:
#
#A       B     C   D       E   F     G    H    I      J       K   L    M    N     O
#657     657   1   2.054   1   657   983   0.0  2.054  2.054   1   0    13   1     2.3
#658     658   1   0.348   0   658   984   0.0  0.348  0.348   0   0    8    1    -2.7
#659     659   1   2.500   0   659   985   0.0  9.300  9.300   1   0   -21   1    -0.6
#659.1   659   2   2.500   0   659   985   0.0  9.300  9.300   1   0   -21   1    -0.6
#659.2   659   3   2.500   0   659   985   0.0  9.300  9.300   1   0   -21   1    -0.6
#659.3   659   4   1.800   1   659   985   0.0  9.300  9.300   1   0   -21   1    -0.6
#
#A = Expanded Unique ID - sequential ID allocated by ds.lexis combined with indicator of
#interval to which that row corresponds. No indicator = interval 1, .N = N+1th interval (e.g.
#see subject 659).
#B = Sequential ID allocated by ds.lexis - starts from ID 1 in each study
#C = Numbered value for each separate time period allocated by ds.lexis. THIS MUST BE CONVERTED
#    INTO A FACTOR AND THEN USED (IN THAT FACTOR FORM NOT AS A NUMERIC) AS A COVARIATE IN THE
#     ds.glm() CALL THAT FITS THE PIECEWISE EXPONENTIAL REGRESSION MODEL.
#D = Exposure time in corresponding interval (e.g. see subject 659). THIS IS THE SURVIVAL TIME
#    VARIABLE TO BE USED IN THE PIECEWISE EXPONENTIAL REGRESSION MODEL - IT IS CONVERTED TO LOG
#    SURVIVAL TIME (LOG TO BASE e) AND USED AS THE OFFSET IN THE ds.glm() MODEL THAT FITS THE
#     PIECEWISE EXPONENTIAL REGRESSION MODEL.
#E = Failure/censoring status in corresponding interval (e.g. see subject 659). THIS IS THE
#    CENSORING VARIABLE TO BE USED IN THE PIECEWISE EXPONENTIAL REGRESSION MODEL - IT IS USED AS
#    THE OUTCOME VARIABLE OF THE ds.glm() MODEL THAT FITS THE PIECEWISE EXPONENTIAL REGRESSION
#    MODEL.
#F = Repeat of Sequential ID allocated by ds.lexis.
#G = Original ID allocated by Opal (note not necessarily in same order as Sequential ID).
#H = starttime (if specified) in original data (note gets repeated across all intervals for any
#    individual). FOR INFORMATION ONLY, DO NOT USE IN PIECEWISE EXPONENTIAL REGRESSION MODEL.
#I = endtime (if specified) in original data (note gets repeated across all intervals for any
#    individual). FOR INFORMATION ONLY, DO NOT USE IN PIECEWISE EXPONENTIAL REGRESSION MODEL.
#J = Total survival time (note gets repeated across all intervals for any individual). FOR
#     INFORMATION ONLY, DO NOT USE IN PIECEWISE EXPONENTIAL REGRESSION MODEL.
#K = Final failure/censoring status (note gets repeated across all intervals for any individual).
#     FOR INFORMATION ONLY, DO NOT USE IN PIECEWISE EXPONENTIAL REGRESSION MODEL.
#L = Additional variable carried into expanded dataframe. In this case the variable is sex
#    (0=male). note repeated across all intervals for any individual.
#M = Additional variable carried into expanded dataframe. note repeated across all intervals for
#     any individual.
#N = Additional variable carried into expanded dataframe. note repeated across all intervals for
#     any individual.
#O = Additional variable carried into expanded dataframe. note repeated across all intervals for
#     any individual.

#EXAMPLE 2
#In this example, the survival time intervals are to be 0<t<=1; 1<t<=2.0, 2.0<t<=5.0, 5.0<t<=11.0,
#then 11.0<t<=maximum survival time in any study (if no survival time exceeds 11, the fifth
#interval will not appear in any row). No expandDF is specified so the output dataframe will be
#named 'CM_expanded'.
#ds.lexis.5(data = "CM", intervalWidth = c(1,1,3,6), idCol = "CM$ID",
```

```
#  entryCol = "CM$STARTTIME", exitCol = "CM$ENDTIME", statusCol = "CM$CENS")
#ds.ls() #to confirm expanded dataframe created

## End(Not run)
```

---

ds.list *Function to construct a list object*

Description

This is similar to the R function `list`.

Usage

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

Arguments

x	a character, the names of the objects to coerce into a list.
newobj	the name of the output object. If this argument is set to NULL, the name of the new object is 'newlist'.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

If the objects to coerce into a list are for example vectors held in a matrix or a dataframe the names of the elements in the list are the names of columns.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Gaye, A.; Isaeva, J.

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")
 opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)
```

```
# combine the 'LAB_TSC' and 'LAB_HDL' variables into a list
myobjects <- c('D$LAB_TSC', 'D$LAB_HDL')
ds.list(x=myobjects)

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


## End(Not run)
```

---

ds.listClientsideFunctions *ds.listClientsideFunctions calling no server-side functions*

Description

Lists all current client-side functions

Usage

```
ds.listClientsideFunctions()
```

Details

Depending on choice of arguments can list all client-side functions or any combination of: dsBaseClient, dsGraphicsClient, dsModellingClient, dsStatsClient, or userDefinedClient. Operates by directly interrogating the R objects stored in the input client packages and objects with name "ds. ....." in .GlobalEnv

Value

list containing all functions in each or all of these five classes

Author(s)

Paul Burton for DataSHIELD Development Team

ds.listDisclosureSettings *ds.listDisclosureSettings*

Description

Lists current values for disclosure control filters in all Opal servers

Usage

```
ds.listDisclosureSettings(datasources = NULL)
```

Arguments

datasources	a list of the particular Opal servers to have their values listed

Details

This function lists out the current values of the eight disclosure filters in each of the Opal servers specified by datasources. Eight filters can currently be set: (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 round 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 0.37 (its default value) the maximum allowable number of parameters in a model fitted to those data will be 370. This disclosure filter protects against fitting overly saturated models which can be disclosive. The choice of 0.37 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.stringShort 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 absense 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 0.10

which means that the noise added to a given variable, follows a normal distribution with zero mean and variance equal to 10 variance of the given variable. Any value greated than this threshold can reduce the risk of disclosure.

Value

a list containing the current settings of the nfilters in each study specified

Author(s)

Paul Burton, Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:
#WORKING EXAMPLES NOT PROVIDED (MULTIPLE OPALS AND R SESSIONS MAKE THIS DIFFICULT)
##Client-side function call to list current disclosure settings in all Opal servers
#ds.listDisclosureSettings()
#
##Equivalent call directly to server-side function to list current disclosure settings in all
##Opal servers not recommended unless you are experienced DataSHIELD user
#ds.look("listDisclosureSettingsDS()")
#
##Call to client-side function and save output as an R object to refer to later
#current.DisclosureSettings<-ds.listDisclosureSettings()
##Interrogate ouput later
#current.DisclosureSettings[[1]]
#
#Restrict call to list disclosure settings only to the first, or second Opals
#ds.listDisclosureSettings(datasources=opals.em[1])
#ds.listDisclosureSettings(datasources=opals.em[2])
#

## End(Not run)
```

---

ds.listServersideFunctions

ds.listServersideFunctions calling no server-side functions

Description

Lists all current server-side functions

Usage

```
ds.listServersideFunctions(datasources = NULL)
```

Arguments

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)]

Details

Uses dsadmin.get_methods function from opaladmin package to list all assign and aggregate functions on the available opal 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.

Value

list containing all serverside functions by study. Firstly lists assign and then aggregate functions.

Author(s)

Burton, PR.

---

ds.log *Computes logarithms, by default natural logarithms*

Description

This function is similar to R function log.

Usage

```
ds.log(x = NULL, base = exp(1), newobj = NULL, datasources = NULL)
```

Arguments

x	a vector.
base	a numrical, the base with respect to which logarithms are computed.
newobj	the name of the new variable. If this argument is set to NULL, the name of the new variable is the name of the input variable with the suffixe '_log' (e.g. 'LAB_TSC_log', if input variable's name is 'LAB_TSC')
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

this is simply a wrapper for the server side function.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Gaye, A.; Isaeva, J.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  myvar <- list("LAB_TSC")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Compute natural logarithm of LAB_TSC
  ds.log(x='D$LAB_TSC')

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


## End(Not run)
```

`ds.look` *ds.look*

Description

The function is a wrapper for the 'opal' package function 'datashield.aggregate'.

Usage

```
ds.look(toAggregate = NULL, checks = FALSE, datasources = NULL)
```

Arguments

`toAggregate`	a character string specifying the function call to be made(see above)

checks — a Boolean object indicating whether optional checks are to be undertaken in this instance only one check is specified - to check that a "toAggregate" expression has been specified, nevertheless this is defaulted to FALSE to be consistent with other functions and to save a small amount of time.

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)]

Details

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. 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 serverside function would usually expect from its clientside 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 clientside function you write ds.table1D('female') the inverted commas are stripped off during processing by the clientside function so the call to the serverside does not contain inverted commas. Apart from during development work (e.g. before a clientside function has been written) it is almost always easier and less error prone to call a serverside function using its client-side pair.

Value

the output from the specified server side aggregate function to the client side

Author(s)

Gaye, A, Burton PR.

---

ds.ls *Returns a vector of character strings giving the names of the objects on remote server*

Description

this function is similar to R function `ls`

Usage

```
ds.ls(datasources = NULL)
```

Arguments

datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

When running analyses one may want to know the objects already generated. This function is not disclosive as it only returns the names of the objects and not their contents. As a restriction only objects in the current environment can be displayed and hidden objects names (names starting with '.') cannot be returned to avoid displaying anything else than the objects generated by the user in the current environment. Unlike the R base version of this function, the DataSHIELD version does not allow to specify another environment, only names of objects on the 'current' environment can be displayed.

Value

a vector of character strings giving the names of the objects in the 'current' environment of the remote opal server.

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")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # assign 'LAB_TSC' in the dataframe D to a new variable 'labtsc'
  ds.assign(toAssign='D$LAB_TSC', newobj='labtsc')

  # now call the ds.ls function to display the names of the objects on the server site
  ds.ls()

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


## End(Not run)
```

ds.make *ds.make*

Description

Makes (calculates) a new object in the R environment on the server side. ds.make is equivalent to ds.assign, but runs slightly faster. It defines a datashield object via an allowed function or an arithmetic expression hence creating a new object in the server side R environments. The function is a wrapper for the 'opal' package function 'datashield.assign'.

Usage

```
ds.make(toAssign = NULL, newobj = "newObject", datasources = NULL)
```

Arguments

toAssign	a character string specifying the function call or the arithmetic expression that generates the newObject. In general the string should be reasonably simple to avoid blocking by the parser and complex (many brackets) expressions can always be broken down into a series of simple steps - e.g. see example 1 below. If toAssign is a simple pre-existing data object, it will simply be copied and assigned as having a second name as specified by the newobject argument - e.g. see example 1 below. One bug identified
newobj	the name of the new object
datasources	specifies the particular opal object(s) to use, if it is not specified the default set of opals will be used. The default opals are always called default.opals. This parameter is set without inverted commas: e.g. datasources=opals.em or datasources=default.opals If you wish to specify the second opal server in a set of three, the parameter is specified: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set specify: e.g. datasources=opals.em[2,3]

Details

If no newobj name is provided, the new object is named 'newObject' by default, otherwise the name can be specified using the newobj argument. If the newObject 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 obviously 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('5.3 + beta*xvar', '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. In order 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('indID-indID+1','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('ONES*5.3','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('vect5.3+beta*xvar', 'predvals'). Alternatively, if you simply swap the original call around: ds.make('(beta*xvar)+5.3', '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 obviously 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('indID-indID','ZEROS') to create a vector of zeros of that same length may also be useful.

Value

the object specified by the newobj argument (or default name newObject) is written to the serverside and a validity message indicating whether the newobject has been correctly created at each source is returned to the client. If it has not been correctly created the return object return.info details in which source the problem exists and whether: (a) the object exists at all; (b) it has meaningful content indicated by a valid class.

Author(s)

DataSHIELD Development Team

Examples

```
## Not run:

##EXAMPLE 1
##CONVERT PROPORTIONS IN prop.rand TO log(odds) IN logodds.rand
#ds.make("(prop.rand)/(1-prop.rand)","odds.rand")
#ds.make("log(odds.rand)","logodds.rand")



##EXAMPLE 2
##MISCELLANEOUS ARITHMETIC OPERATORS: ARBITRARY CALCULATION
##USE DEFAULT NEW OBJECT NAME
#ds.make("((age.60+bmi.26)*(noise.56-pm10.16))/3.2")



##EXAMPLE 3
##MISCELLANEOUS OPERATORS WITHIN FUNCTIONS (female.n is binary 1/0 so female.n2 = female.n
##and so they cancel out in code for second call to ds.make and so that call is
##equivalent to copying log.surv to output.test.1)
```

```
#ds.make("female.n^2","female.n2")
#ds.make("(2*female.n)+(log.surv)-(female.n2*2)","output.test.1")
#ds.make("exp(output.test.1)","output.test")

## End(Not run)
```

---

ds.matrix *ds.matrix calling assign function matrixDS*

Description

Creates a matrix A on the serverside

Usage

```
ds.matrix(mdata = NA, from = "clientside.scalar",
  nrows.scalar = NULL, ncols.scalar = NULL, byrow = FALSE,
  dimnames = NULL, newobj = NULL, datasources = NULL)
```

Arguments

mdata	specifies the elements of the matrix to be created. If it is a vector it should usually be the same length as the total number of elements in the matrix and these will fill the matrix column by column or row by row depending whether the argument <byrow> is FALSE (default) or TRUE. If the mdata vector is not the same length as the total number of elements in the matrix to be created, the values in mdata will be used repeatedly until all elements in the matrix are full. If mdata is a scalar, all elements in the matrix will take that value. The <mdata> argument can be specified either as a character string specifying the name of a serverside scalar or vector, or a numeric value (or numeric object) representing a scalar specified from the clientside. Zeros, negative values and NAs are all allowed.
from	a character string specifying the source and nature of <mdata>. Can only take values "serverside.vector", "serverside.scalar" or "clientside.scalar" This argument must be specified - NULL values not permitted. Defaults to "clientside.scalar"
nrows.scalar	specifies the number of rows in the matrix to be created. It can be a character string specifying the name of a serverside scalar: e.g. if a serverside scalar named "ss.scalar" exists and holds the value 23, then by specifying nrows.scalar="ss.scalar", the matrix to be created will have 23 rows. Alternatively it can be specified as a numeric value (or numeric object) representing a scalar specified from the clientside: e.g. nrows.scalar=14 or equivalently scalar.value<- 14; nrows.scalar=scalar.value. This will create a matrix with 14 rows. A zero, negative, NULL or missing value is not permitted

ncols.scalar	specifies the number of columns in the matrix to be created. It can be a character string specifying the name of a serverside scalar: e.g. if a serverside scalar named "ss.scalar" exists and holds the value 23, then by specifying ncols.scalar="ss.scalar", the matrix to be created will have 23 columns. Alternatively it can be specified as a numeric value (or numeric object) representing a scalar specified from the clientside: e.g. ncols.scalar=14 or equivalently scalar.value<- 14; ncols.scalar=scalar.value. This will create a matrix with 14 columns. A zero, negative, NULL or missing value is not permitted
byrow	a logical value specifying whether, when <mdata> is a vector, the matrix created should be filled row by row (byrow=TRUE) i.e. starting at the first element of first row, filling that row and then moving to the first element of the second row and filling that row etc until all elements of the matrix are full or column by column (byrow=FALSE). Default = FALSE.
dimnames	A dimnames attribute for the matrix: NULL or a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names only.
newobj	A character string specifying the name of the matrix to which the output is to be written. If no <newobj> argument is specified or it is NULL the output matrix names defaults to "new_matrix"
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)]

Details

Similar to the matrix() function in native R. Creates a matrix with dimensions specified by <nrows.scalar> and <ncols.scalar> arguments and assigns the values of all its elements based on the <mdata> argument

Value

the object specified by the <newobj> argument (or default name "new_matrix") 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.matrix also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

Examples

```
## Not run:

ds.matrix(mdata=-13,from="clientside.scalar", nrows.scalar=3,ncols.scalar=8,newobj="cs.block")

ds.matrix(NA,from="clientside.scalar", nrows.scalar=4,ncols.scalar=5,newobj="cs.block.NA")

clientside.input.scalar<-837
ds.matrix(clientside.input.scalar,from="clientside.scalar", nrows.scalar=7,ncols.scalar=3,
          newobj="cs.block.input")

clientside.input.nrows.scalar<-11
ds.matrix(clientside.input.scalar,from="clientside.scalar",
          nrows.scalar=clientside.input.nrows.scalar,ncols.scalar=3,
          newobj="cs.block.input.nrows")

ds.rUnif(samp.size=1,min=0.5,max=10.5,newobj="block.scalar",seed.as.integer=761728,
           force.output.to.k.decimal.places = 0)

ds.matrix("block.scalar",from="serverside.scalar", nrows.scalar=9,ncols.scalar=7,
          newobj="ss.block")

ds.make("log(block.scalar*(-1))","block.scalar.NA")

ds.matrix("block.scalar.NA",from="serverside.scalar", nrows.scalar=9,ncols.scalar=7,
          newobj="ss.block.NA")

ds.matrix("block.scalar",from="serverside.scalar", nrows.scalar="block.scalar",
          ncols.scalar="block.scalar",newobj="ss.block.square")

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)
ds.matrix("ss.vector",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,
          newobj="sv.block")

ds.rUnif(samp.size=5,min=-10.5,max=10.5,newobj="ss.vector.5",seed.as.integer=551625,
           force.output.to.k.decimal.places = 0)
ds.matrix("ss.vector.5",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,
          newobj="sv.block.5")

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)
ds.matrix("ss.vector.9",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,byrow=TRUE,
          newobj="sv.block.9")

ds.matrix("ss.vector.9",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,
          newobj="sv.block.9.ragged")
```

```
ds.rUnif(samp.size=12,min=-10.5,max=10.5,newobj="ss.vector.12",seed.as.integer=778172,
           force.output.to.k.decimal.places = 0)

ds.matrix("ss.vector.12",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,
          newobj="sv.block.12.ragged")

ds.recodeValues("ss.vector", c(-10),c(NA),newobj="ss.vector.NA")
ds.matrix("ss.vector.NA",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,
          newobj="sv.block.NA")

ds.matrix("ss.vector.NA",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,
          byrow=TRUE,newobj="sv.byrow.block")

ds.matrix(NA, nrows.scalar=7,ncols.scalar=6,newobj="empty.matrix")

ds.matrix("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")

ds.matrix("ss.vector.9",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,byrow=TRUE,
dimnames=list(c("a","b","c","d","e"),c(10*(9:1))),newobj="sv.block.9.dimnames12")

#No specification of newobj
ds.matrix("ss.vector.9",from="serverside.vector", nrows.scalar=5,ncols.scalar=9,byrow=TRUE,
dimnames=list(c("a","b","c","d","e"),c(10*(9:1))))

## End(Not run)
```

---

ds.matrixDet *ds.matrixDet calling assign function matrixDetDS2*

Description

Calculates the determinant of a square matrix A and writes it as a data object to the serverside

Usage

```
ds.matrixDet(M1 = NULL, newobj = NULL, logarithm = FALSE,
  datasources = NULL)
```

Arguments

M1	A character string specifying the name of the matrix for which determinant to be calculated
newobj	A character string specifying the name of the matrix to which the output is to be written. If no <newobj> argument is specified, the output matrix names defaults to "M1_det" where <M1> is the first argument of the function
logarithm	logical. Default is FALSE, which returns the determinant itself, TRUE returns the logarithm of the modulus of the determinant.

`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)]

Details

Calculates the determinant of a square matrix (for additional information see help for determinant function in native R). This operation is only possible if the number of columns and rows of A are the same.

Value

the calculated determinant of the matrix as a serverside object with its name specified by the <newobj> argument (or default name <M1>_det). The determinant is reported as a two component list. Element 1 is $modulus and element 2 is $sign. If logarithm=FALSE: $modulus reports the absolute value of the determinant and is therefore always positive. $sign indicates whether the determinant is positive ($sign=1) or negative ($sign=-1). $modulus has an attribute [attr(,"logarithm")] which is FALSE if the argument <logarithm> was FALSE - this enables you to look at results post-hoc to determine whether the logarithm argument was TRUE or FALSE. If you wish to generate the actual determinant if logarithm=FALSE it is easiest to calculate $modulus*$sign. If logarithm=TRUE: $modulus reports the log (to base e) of the absolute value of the determinant. $sign again reports whether the determinant is positive ($sign=1) or negative ($sign=-1). The attribute of $modulus [attr(,"logarithm")] is now TRUE. If you wish to generate the actual determinant when logarithm=TRUE you calculate exp($modulus)*$sign. In addition to the calculated matrix determinant, two validity messages are also 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.matrixDet also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

ds.matrixDet.report *ds.matrixDet.report calling aggregate function matrixDetDS1*

Description

Calculates the determinant of a square matrix A and returns the result to the clientside

Usage

```
ds.matrixDet.report(M1 = NULL, logarithm = FALSE, datasources = NULL)
```

Arguments

M1	A character string specifying the name of the matrix for which determinant to be calculated
logarithm	logical. Default is FALSE, which returns the determinant itself, TRUE returns the logarithm of the modulus of the determinant.
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)]

Details

Calculates the determinant of a square matrix (for additional information see help for determinant function in native R). This operation is only possible if the number of columns and rows of A are the same.

Value

the matrix determinant for A to the clientside. Calculated separately for each study. The determinant is reported as a two component list. Element 1 is $modulus and element 2 is $sign. If logarithm=FALSE: $modulus reports the absolute value of the determinant and is therefore always positive. $sign indicates whether the determinant is positive ($sign=1) or negative ($sign=-1). $modulus has an attribute [attr(,"logarithm")] which is FALSE if the argument <logarithm> was FALSE - this enables you to look at results post-hoc to determine whether the logarithm argument was TRUE or FALSE. If you wish to generate the actual determinant if logarithm=FALSE it is easiest to calculate $modulus*$sign. If logarithm=TRUE: $modulus reports the log (to base e) of the absolute value of the determinant. $sign again reports whether the determinant is positive ($sign=1) or negative ($sign=-1). The attribute of $modulus [attr(,"logarithm")] is now TRUE. If you wish to generate the actual determinant when logarithm=TRUE you calculate exp($modulus)*$sign. If the function fails in any study for a reason which is identified, an explanatory error message is returned instead of the object containing the calculated matrix determinant

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.matrixDiag` *ds.matrixDiag calling assign function matrixDiagDS*

Description

Extracts the diagonal vector from a square matrix A or creates a diagonal matrix A based on a vector or a scalar value and writes the output to the serverside

Usage

```
ds.matrixDiag(x1 = NULL, aim = NULL, nrows.scalar = NULL,
  newobj = NULL, datasources = NULL)
```

Arguments

`x1`	This argument determines the input object. Depending on the specified value of the <aim> argument it may be a character string identifying the name of a serverside vector or scalar. Alternatively, it may be a single number e.g. 29, or a vector specified as e.g. c(3,5,-2,8) or it can be the name (but not in inverted commas) of a clientside scalar or clientside vector which have already been assigned values e.g. scalar.s<-83, x1=scalar.s; or vector.v<-c(7,0,-2,3:9), x1=vector.v.
`aim`	a character string specifying what behaviour is required of the function. It must take one of five values: "serverside.vector.2.matrix"; "serverside.scalar.2.matrix"; "serverside.matrix.2.vector"; "clientside.vector.2.matrix"; or "clientside.scalar.2.matrix"
`nrows.scalar`	if this takes value k, it forces the output matrix to have k rows and k columns. If x1 is a scalar, this argument must be set as otherwise the dimensions of the square matrix are undefined. If x1 is a vector and no value is set for the <nrows.scalar> argument the dimensions of the square matrix are defined by the length of the vector. If x1 is a vector and the <nrows.scalar> is set at k, the vector will be used repeatedly to fill up the diagonal. If for example the vector is of length 7 and <nrows.scalar> is specified as 18, a square diagonal matrix with 18 rows and 18 columns will be created with the vector repeated twice from element 1,1 to element 14,14 and the first 4 elements of the vector will fill diagonal elements 15,15 to 18,18. All off diagonal elements will be 0. The <nrows.scalar> argument can either be set as a number, a clientside scalar holding a single number or a character string representing the name of a serverside scalar.
`newobj`	A character string specifying the name of the output object to be written to the serverside which may be a matrix or a vector depending on the value of the <aim> argument.If no <newobj> argument is specified, the output object name defaults to "diag".

`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)]

Details

Depending on the specified value of the <aim> argument this function behaves differently. If aim=="serverside.vector.2.matrix" the function takes a serverside 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 is of length k, the output matrix has k rows and k columns. If aim=="serverside.scalar.2.matrix" the function takes a serverside 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; If aim=="serverside.matrix.2.vector" the function takes a square serverside matrix and extracts its diagonal values as a vector which is written to the serverside; If aim=="clientside.vector.2.matrix" the function takes a vector specified on the clientside and writes out a square matrix to the serverside 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 is of length k, the output matrix has k rows and k columns; If aim=="clientside.scalar.2.matrix" the function takes a scalar specified on the clientside 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.

Value

the matrix or vector specified by the <newobj> argument (or default name diag) 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.matrixDiag also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

ds.matrixDimnames *ds.matrixDimnames calling assign function matrixDimnamesDS*

Description

Adds dimnames (row names, column names or both) to a matrix on the serverside.

Usage

```
ds.matrixDimnames(M1 = NULL, dimnames = NULL, newobj = NULL,
  datasources = NULL)
```

Arguments

M1	Specifies the name of the serverside matrix to which dimnames are to be added. Specified as a character string in inverted commas: e.g. M1="matrix.name"
dimnames	A dimnames attribute for the matrix: NULL or a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names only. Components in the list can be specified as character strings (e.g. "a", "name3" or "77"), or numbers (e.g. 1,-8, 1:10). Examples include: dimnames=list(c("a","cc","73",8,"h"),c("1","b","d","8","ghhj",1:4)) specifies the row names and column names for a matrix with 5 rows and 9 columns; dimnames=list(NULL,c("1","b","d","8","ghhj",1:4)) specifies just the column names for a matrix with 9 columns; dimnames=list(c("a","cc","73",8,"h"),NULL) specifies just the row names for a matrix with 5 rows.
newobj	A character string specifying the name of the matrix to which the output is to be written. If no <newobj> argument is specified or it is NULL the output matrix names defaults to "<M1>_dimnames" where <M1> is the matrix name specified by the <M1> argument
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)]

Details

Adds dimnames (row names, column names or both) to a matrix on the serverside. Similar to the dimnames function in native R.

Value

the object specified by the <newobj> argument (or default name "<M1>_dimnames") 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.matrixDimnames also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.matrixInvert` *ds.matrixInvert calling assign function matrixInvertDS*

Description

Inverts a square matrix A and writes the output to the serverside

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 specifying the name of the matrix to which the output is to be written. If no <newobj> argument is specified, the output matrix names defaults to "M1_inverted" where <M1> is the first argument of the function
`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)]

Details

Undertakes standard matrix inversion. This operation is only possible if the number of columns and rows of A are the same and the matrix is non-singular - positive definite (eg there is no row or column that is all zeros)

Value

the object specified by the <newobj> argument (or default name <M1>_inverted) 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.matrixInvert also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

---

ds.matrixMult *ds.matrixMult calling assign function matrixMultDS*

Description

Calculates the matrix product of two matrices and writes output to serverside

Usage

```
ds.matrixMult(M1 = NULL, M2 = NULL, newobj = NULL,
  datasources = NULL)
```

Arguments

M1	A character string specifying the name of the first matrix (M1)
M2	A character string specifying the name of the second matrix (M2)
newobj	A character string specifying the name of the matrix to which the output is to be written. If no <newobj> argument is specified, the output matrix names defaults to "M1_M2" where <M1> is the first argument of the function and <M2> is the second argument of the function.

`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)]

Details

Undertakes standard matrix multiplication where with input matrices A and B with dimensions A: mxn and B: nxp the output C has dimensions mxp and each elemnt C[i,j] has value equal to the dot product of row i of A and column j of B where the dot product is obtained as sum(A[i,1]*B[1,j] + A[i,2]*B[2,j] + ... + A[i,n]*B[n,j]). This calculation is only valid if the number of columns of A is the same as the number of rows of B

Value

the object specified by the <newobj> argument (or default name <M1>_<M2>) 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.matrixMult also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

---

ds.matrixTranspose *ds.matrixTranspose calling assign function matrixTransposeDS*

Description

Transposes a matrix A and writes the output to the serverside

Usage

```
ds.matrixTranspose(M1 = NULL, newobj = NULL, datasources = NULL)
```

Arguments

M1	A character string specifying the name of the matrix to be transposed
newobj	A character string specifying the name of the matrix to which the output is to be written. If no <newobj> argument is specified, the output matrix names defaults to "M1_transposed" where <M1> is the first argument of the function
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)]

Details

Undertakes standard matrix transposition. 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.

Value

the object specified by the <newobj> argument (or default name <M1>_transposed) 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.matrixInvert also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

---

ds.mean *Computes the statistical mean of a given vector*

Description

This function is similar to the R function mean.

Usage

```
ds.mean(x = NULL, type = "split", checks = FALSE,
  save.mean.Nvalid = FALSE, datasources = NULL)
```

Arguments

x	a character, typically the name of a numerical vector
type	a character which represents the type of analysis to carry out. If type is set to 'combine', 'combined', 'combines' or 'c', a global mean is calculated if type is set to 'split', 'splits' or 's', the mean is calculated separately for each study. if type is set to 'both' or 'b', both sets of outputs are produced
checks	a Boolean indicator of whether to undertake optional checks of model components. Defaults to checks=FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed
save.mean.Nvalid	a Boolean indicator of whether the user wishes to save the generated values of the mean and of the number of valid (non-missing) observations into the R environments at each of the data servers. Will save study-specific means and Nvalids as well as the global equivalents across all studies combined. 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.
datasources	specifies the particular opal object(s) to use, if it is not specified the default set of opals will be used. The default opals are always called default.opals. This parameter is set without inverted commas: e.g. datasources=opals.em or datasources=default.opals If you wish to specify the second opal server in a set of three, the parameter is specified: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set specify: e.g. datasources=opals.em[2,3]

Details

It is a wrapper for the server side function.

Value

a list including: Mean.by.Study = estimated mean in each study separately (if type = split or both), with Nmissing (number of missing observations), Nvalid (number of valid observations), Ntotal (sum of missing and valid observations) also reported separately for each study; Global.Mean = Mean, Nmissing, Nvalid, Ntotal across all studies combined (if type = combine or both); Nstudies = number of studies being analysed; ValidityMessage indicates whether a full analysis was possible or whether one or more studies had fewer valid observations than the nfilter threshold for the minimum cell size in a contingency table. If save.mean.Nvalid=TRUE, ds.mean writes the objects "Nvalid.all.studies", "Nvalid.study.specific", "mean.all.studies", and "mean.study.specific" to the serverside on each server

Author(s)

Burton PR; Gaye A; Isaeva I;

See Also

ds.quantileMean to compute quantiles.

ds.summary to generate the summary of a variable.

Examples

```
## Not run:

#  # load that contains the login details
#  data(logindata)
#  library(opal)
#
#  # login and assign specific variable(s)
#  myvar <- list('LAB_TSC')
#  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)
#
#  # Example 1: compute the pooled statistical mean of the variable 'LAB_TSC' - default behaviour
#  ds.mean(x='D$LAB_TSC')
#
#  # Example 2: compute the statistical mean of each study separately
#  ds.mean(x='D$LAB_TSC', type='split')
#
#  # clear the Datashield R sessions and logout
#  datashield.logout(opals)


## End(Not run)
```

---

ds.meanByClass *Computes the mean and standard deviation across categories*

Description

This function calculates the mean and 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, the name of the dataset to get the subsets from or a text formula of the form 'A~B' where A is a single continuous vector and B a single factor vector
outvar	a character vector, the names of the continuous variables
covar	a character vector, the names of up to 3 categorical variables
type	a character which represents the type of analysis to carry out. If type is set to 'combine', a pooled table of results is generated. If type is set to 'split', a table of results is genrated for each study.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

The functions 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 'split' 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 really interested in but including only those studies in the parameter 'datasources'.

Value

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)

Gaye, A.

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:

  # load that contains the login details
  data(logindata)

 # Example 1: calculate the pooled mean proportion for LAB_HDL across GENDER categories where
  # both vectors are in a tabe structure "D"
  # login and assign LAB_HDL and GENDER to a table "D"
 opals <- datashield.login(logins=logindata,assign=TRUE, variables=list('LAB_HDL', 'GENDER'))
  ds.meanByClass(x='D$LAB_HDL~D$GENDER')
```

```
# Example 2: calculate the mean proportion for LAB_HDL across GENDER categories where both
# vectors are 'loose' (i.e. not in a table)
# assign both LAB_HDL and GENDER to vectors not held in a table
ds.assign("D$LAB_HDL", "ldl")
ds.assign("D$GENDER", "sex")
ds.meanByClass(x='ldl~sex')
datashield.logout(opals)

# Example 3: calculate the mean proportion for LAB_HDL across gender, bmi and diabetes status
# categories login and assign all the variables stored on opal
opals <- datashield.login(logins=logindata,assign=TRUE)
ds.meanByClass(x='D', outvar=c('LAB_HDL','LAB_TSC'), covar=c('GENDER','PM_BMI_CATEGORICAL',
               'DIS_DIAB'))

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


## 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`	This must be named as a character string (e.g. "AGE"). It must denote a continuous variable of class numeric.
`y`	This must be named as a character string (e.g. "sex"). It must denote a categorical variable of class factor.
`type`	This must be specified as a character string ("combine", "split" or "both"). If "combine" the results for each group are reported combined over all studies. If "split" the table of means by group (for example) has a separate column for each study. If "both" the tables each have an additional column reporting the sum across all studies in each group in addition to the columns (produced by "split") for each study alone. This parameter defaults to "both" unless "combine" or "split" are specified.

`do.checks` This parameter defaults to FALSE. It determines whether 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. By defaulting the checks to FALSE, we save time, and if you hit a problem that you cannot understand you can reset do.checks to TRUE and make sure that the input objects are correctly defined in all studies.

`datasources` a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources. If no datasources are specified, DataSHIELD looks for opal objects. But, beware, if you have several different sets of opal objects in your analysis space, if you do not explicitly specify which ones you want, you may be asked to state which ones you want after every command or you may get a repeated warning about it after every command. In general, you will typically only need one set of opal objects in any one setting and so this problem will not arise. The datasources argument also allows you to restrict analysis to one or more specific studies rather than all studies. For example, if the Opal objects are called Opal.servers and there are five of them, datasources=Opal.servers[3], will restrict the analysis to the server listed third in your list of servers.

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. It also reports the total number of observations (=Ntotal), the total number of valid observations, i.e. non-missing observations = observations where neither the continuous variable nor the factor is misssing, (Nvalid) and the total number of missing observations (Nmissing). Nvalid=Ntotal-Nmissing and all three quantities represent the sum across all groups and all studies. If any one subgroup consists of between 1 and "nfilter" observations, the function simply reports that fact and suggests that you use a different grouping variable. As in other functions such as ds.table1D, the value of nfilter can be chosen by the data custodian when each Opal server is originally set up. By default it is set to 5. There are IMPORTANT DIFFERENCES between ds.meanSdGp 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 and 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: sex=0, BMI=1, newfactor=1; sex=0, BMI=2, newfactor=2; sex=0, BMI=3, newfactor=3; sex=1, BMI=1, newfactor=4; ds.assign command and you then use newfactor as the single categorising factor. (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 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.

Value

If type = "combine" the function returns a list consisting of a four tables denoting: mean by group; standard deviation (SD) by group; number of non-missing observations (Nvalid) by group; and standard error of the mean (SEM) by group. All of these are COMBINED ACROSS STUDIES. For information, SEM = SD/sqrt(Nvalid). These are all returned in list format with names: Mean_gp, StDev_gp, Nvalid_gp and SEM_gp. If you need to use them in their original class (e.g. matrix), you need to use the conventional R function unlist() to convert them back to their original form. The output list also includes: Total_Nvalid (the total number of valid [non-missing] observations across all groups in all studies; Total_Nmissing (the total number of observations with either or both x and y missing); and Total_Ntotal (the total number of observations [with data missing or not]). If type ="split", the mean, SD, Nvalid and SEM are reported by group and by study. The first four elements of the returned output list are therefore: Mean_gp_study; StDev_gp_study; Nvalid_gp_study; and SEM_gp_study. If there are three studies and we are breaking things down by five groups in each study, each of the first four list elements consists of a table with five rows (one for each group) and three columns (one for each study). The returned output also includes Total_Nvalid, Total_Nmissing and Total_Ntotal as before. If type = "both" the output is precisely the same as with type="split" and each of its components has the same name, but each table (e.g. Mean_gp_study) will now have an extra column on the right had side (so a fourth column in the example above) which contains the appropriate combined value in each group across all studies together. In other words, the four columns replicate the results obtained when type = "combine". CRUCIALLY, IF ONE OR MORE OF THE GROUPS IN ANY OF THE STUDIES CONTAINS BETWEEN 1 and nfilter OBSERVATIONS, the returned output list will ONLY include the warning: [1] "At least 1 cell count is 1-nfilter, please regroup".

Author(s)

Burton PR

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:

# #load that contains the login details
# data(logindata)

# #Example 1: Calculate the mean, SD, Nvalid and SEM of the continuous variable AGE.60 (age in
# #years centralised at 60), broken down by TID.f (a six level factor relating to survival time)
```

```
# #and report the pooled results combined across studies.
# ds.meanSdGp("AGE.60","TID.f","combine")

# #Example 2: Calculate the mean, SD, Nvalid and SEM of the continuous variable AGE.60 (age in
# #years centralised at 60), broken down by TID.f (a six level factor relating to survival time)
# #and report both study-specific results and the pooled results combined across studies. Do the
# #checks for consistency of variables in all studies. Save the returned output to msg.b.
# msg.b <- ds.meanSdGp("AGE.60", "SEXF", "both", do.checks=TRUE)
# msg.b
# PRODUCES THIS OUTPUT
# $Mean_gp_study
#           study1    study2   COMBINE
# SEXF_1 -4.099893 -5.199134 -4.568966
# SEXF_2 -2.384477 -3.057421 -2.690300
#
# $StDev_gp_study
#          study1   study2  COMBINE
# SEXF_1 13.67313 14.52537 14.04313
# SEXF_2 14.87182 14.64741 14.77026
#
# $Nvalid_gp_study
#        study1 study2 COMBINE
# SEXF_1    931    693    1624
# SEXF_2   1108    923    2031
#
# $SEM_gp_study
#           study1    study2   COMBINE
# SEXF_1 0.4481188 0.5517731 0.3484744
# SEXF_2 0.4467804 0.4821253 0.3277427
#
# $Total_Nvalid
# [1] 3655
#
# $Total_Nmissing
# [1] 45
#
# $Total_Ntotal
# [1] 3700
#
# Example 3:
# Calculate the mean, SD, Nvalid and SEM of the continuous variable SBP (systolic BP), broken
# down by CVA (1 = had a stroke, 0 = no stroke) report the study-specific results only. The
# output shows that there are inadequate numbers of stroke cases to carry out this particular
# analysis: at least one cell contains between 1 and nfilter (the chosen value of the disclosure
# filter - typically 4) observations. Given that the CVA grouping is as simple as it can be, it
# is impossible to regroup in this setting. The only option would be to have chosen a different
# level for the nfilter. If it is 0, there is no limitation on cell counts: but whether or not
# cell counts in the range, say, 1-4 are to be viewed as providing a significant risk is
# something that should be decided before the analysis starts. In reality, for many biomedical
# studies, particularly when data users have signed a data access agreement explicitly stating
# they will not try to identify individuals or infer their characteristics, many researchers may
# choose to turn the disclosure filter off (nfilter=0). But for particularly sensitive data, or
# data obtained from official governmental sources, e.g. census data, there may simply be no
```

```
# option but to pick a filter of say 4.
# ds.meanSdGp("SBP", "CVA", "split")
# PRODUCES THIS OUTPUT
# [1] "At least 1 cell count is 1-nfilter, please regroup"
#
# clear the Datashield R sessions and logout
#datashield.logout(opals)


## End(Not run)
```

---

ds.merge *ds.merge calling assign function mergeDS*

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,	the name of the first data.frame to be merged specified in inverted commas. For example: x.name='dfx.name'. Native R refers to the first data.frame as x and the second as y.
y.name,	the name of the second data.frame to be merged specified in inverted commas. For example: y.name='dfy.name'. Native R refers to the first data.frame as x and the second as y.
by.x.names	the name of a single variable or a vector of names of multiple variables containing the IDs or other data on which data.frame x is to be merged/linked to data.frame y. Names must be specified in inverted commas. For example: by.x.names='individual.ID' or by.x.names=c('year.of.birth', 'month.of.birth', 'day.of.birth', 'surname')
by.y.names	the name of a single variable or a vector of names of multiple variables containing the IDs or other data on which data.frame y is to be merged/linked to data.frame x. Names must be specified in inverted commas. For example: by.y.names='individual.ID' or by.y.names=c('year.of.birth', 'month.of.birth', 'day.of.birth', 'surname')
all.x	logical, if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. Default is FALSE, so that only rows with data from both x and y are included in the output.

all.y	logical, if TRUE, then extra rows will be added to the output, one for each row in y that has no matching row in x. These rows will have NAs in those columns that are usually filled with values from x. Default is FALSE, so that only rows with data from both x and y are included in the output.
sort	logical, if TRUE the merged result should be 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. This avoids ambiguity in the source of columns that are not used for merging. Default is .x for vector names in the first data.frame and .y for vector names in the second data.frame
no.dups	logical, indicating that suffixes are appended in more cases to rigorously avoid duplicated column names in the merged data.frame. Default TRUE but was apparently implicitly FALSE before R version 3.5.0.
incomparables,	values which cannot be matched. See 'match' in help for Native R merge function. This is intended to be used for merging on one column, so these are incomparable values of that column.
newobj	the name of the merged data.frame. If this argument is set to NULL, the name of the merged data.frame is defaulted to 'x.name_y.name' where x.name is the name of the first input data.frame specified as the <x.name> argument and y.name is the name of the second input data.frame specified as the <y.name> argument.
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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

For further information see details of the native R function merge. In choosing which variables to use to merge/link the data.frames the native R 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

Value

the merged data.frame specified by the <newobj> argument (or by default 'x.name_y.name' if the <newobj> argument is NULL) which is written to the serverside. In addition, two validity messages are returned to the clientside 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 there may be a studysideMessage that can explain the error in creating the full output object. As well as appearing

on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message(<newobj>) it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message(<newobj>) will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

`ds.message` *Return a 'studysideMessage' written by an assign function to server-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, containing the name of the list containing the message. As an example, the server-side function mergeDS enacts the command: datashield.assign(datasources, "messageobj", calltext2) As a standard assign function its output is directed to the list object named (in this case) "messageobj". If a studysideMessage is written by DataSHIELD it can be found as messageobj$studysideMessage. To read it, you have to issue the client-side command: ds.message('messageobj'). This tells DataSHIELD to look for the server-side object 'messageobj' and if it finds it, to return any text held in messageobj$studysideMessage. In order to help users to know the name of the server-side list object to ask for in issueing the command: ds.message('messageobj') developers are asked to include a message such as: Note3<-"IF FUNCTION FAILED ON ONE OR MORE STUDIES WITHOUT EXPLANATION, TYPE [PRECISELY] THE COMMAND:" Note4<-"ds.message('messageobj') FOR MORE ERROR MESSAGES" These represent two of four notes returned by the client-side function ds.merge at the end of each call. In combination, these two notes alert the user to the fact that if there is an error, there may be additional information available in a studysideMessage, and also tells them how to retrieve that message.

`datasources`
: specifies the particular opal object(s) to use, if it is not specified the default set of opals will be used. The default opals are always called default.opals. This parameter is set without inverted commas: e.g. datasources=opals.em or

datasources=default.opals If you wish to specify the second opal server in a set of three, the parameter is specified: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set specify: e.g. datasources=opals.em[2,3]

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. ds.message calls messageDS which looks specifically for an object called $serversideMessage in a designated list on the server-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, and as additional protection against unexpected ways that someone may try to get round this limitation, a $studysideMessage is a string that cannot exceed a length of nfilter.string a default of 80 characters.

Value

a list object from each study, containing whatever message has been written by DataSHIELD into $studysideMessage.

Author(s)

Burton PR

---

ds.names *Gets the names of items in a list*

Description

This function is similar to the R function `names`.

Usage

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

Arguments

`x`	a character, the name of an object of type list.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

In DataSHIELD the use of this function is restricted to objects of type list.

Value

The names of the list's elements for each study

Author(s)

Gaye, A.

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")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

 # generates subset tables from the table assigned above (by default the table name is 'D')
  ds.subsetByClass(x='D', subsets='subclasses')

 # the above object 'subsets' is a list, let us display the names of elements in 'subsets'
  ds.names('subclasses')

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


## End(Not run)
```

---

ds.numNA *Gets the number of missing values in a vector*

Description

In DataSHIELD it is not possible to visualize the data. This function helps to know the number of missing values in a vector to eventually use a vector of equal length (i.e. the count of missing entries) to replace the missing values.

Usage

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

Arguments

x	a character, the name of a vector to check for missing entries.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

The vector to check for missing values might be a in a table structure or not. The number of missing entries are counted and the total for each study is returned.

Value

for an array, NULL or a vector of mode integer

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the stored variables.
  opals <- datashield.login(logins=logindata,assign=TRUE)

 # Example 1: Get the number of missing values in the variable 'LAB_HDL' held in the table 'D'.
  ds.numNA(x='D$LAB_HDL')

 # Example 2: Assign the above variable and check the number of missing values on the now loose
  # variable 'LAB_HDL'.
  ds.assign(toAssign='D$LAB_HDL', newobj='labhdl')

  # Example 3: Get the pooled dimension of the assigned datasets
  ds.numNA(x='labhdl')

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


## End(Not run)
```

---

ds.quantileMean *Compute the quantiles*

Description

This function calculate the mean and quantile values of a quantitative variable

Usage

```
ds.quantileMean(x = NULL, type = "combine", datasources = NULL)
```

Arguments

x	a character, the name of the numeric vector.
type	a character which represent the type of graph to display. If type is set to 'combine' pooled values are displayed and sumamries a returned for each study if type is set to 'split'
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

Unlike standard r summary function the minimum and maximum values afe not returned because they are potentially disclosive.

Value

quantiles and statistical mean

Author(s)

Gaye, A.

See Also

ds.mean to compute statistical mean.

ds.summary to generate the summary of a variable.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  myvar <- list('LAB_HDL')
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Example 1: plot a combined histogram of the variable 'LAB_HDL' - default behaviour
  ds.quantileMean(x='D$LAB_HDL')

  # Example 2: Plot the histograms separately (one per study)
  ds.quantileMean(x='D$LAB_HDL', type='split')

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


## End(Not run)
```

ds.rbind *ds.rbind calling rbindDS*

Description

Take a sequence of vector, matrix or data-frame arguments and combine them by row to produce a matrix.

Usage

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

Arguments

x	This is a vector of character strings representing the names of the elemental components to be combined. For example, the call: ds.rbind(x=c('matrix.m','matrix.n'),newobj='rbind_outpu will stack matrix.m on top of matrix.n provided the number of columns of matrix.m and matrix.n are the same. The output object rbind_output is written to the serverside. As many elemental components as needed may be combined using ds.rbind provided they all have the same number of columns. For convenience the x argument can alternatively be specified in a two step procedure, the first being a call to the native R environment on the client server: x.components<-c('matrix.m','matrix.n') then ds.rbind(x=x.components,newobj='rbind_output'). Column names are taken either from the column names of the first object specified in the <x> argument. Alternatively new column names can be user specified using <force.colnames>
DataSHIELD.checks	logical, if TRUE checks are made that all input objects exist and are of an appropriate class. These checks are relatively slow and so the <DataSHIELD.checks> argument is defaulted to FALSE
force.colnames	NULL or a vector of character strings representing the required column names of the output object. For example: force.colnames=c("colname1","name.of.second.column", "lastcol") for an output object with three columns. 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 there are columns in the output object. In other words as every specified object in the <x> argument must have the same number of columns the vector of column names must have the same number of elements as there are columns in every object specified in the <x> argument If the length of the column name vector is incorrect a studysideMessage is returned: "Number of column names does not match number of columns in output object. Here 'N' names are required.Please see help for ds.rbind function" where 'N' is the actual number of columns in the output object
newobj	This a character string providing a name for the output data.frame which defaults to 'cbind.out' 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

A sequence of vector, matrix or data-frame arguments is combined row by row to produce a matrix which is written to the serverside. For more details see the native R function rbind. The handling of argument <x> is similar to that of functions ds.cbind and ds.dataFrame

Value

the object specified by the <newobj> argument (or default name <rbind.out>). which is written to the serverside. Unlike the ds.cbind function even if one of the objects specified in the <x> argument is a data.frame the output object will always be of class matrix As well as writing the output object as <newobj> on the serverside, 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.cbind() also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("<newobj>") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("<newobj>") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.rBinom` *ds.rBinom calling rBinomDS and setSeedDS*

Description

Generates random (pseudorandom) numbers from a binomial distribution

Usage

```
ds.rBinom(samp.size = 1, size = 0, prob = 1, newobj = "newObject",
  seed.as.integer = NULL, return.full.seed.as.set = FALSE,
  datasources = NULL)
```

Arguments

samp.size	the length of the random number vector to be created in each source. <samp.size> can be a numeric scalar and this then specifies the length of the random vectors in each source to be the same. If it is a numeric vector it enables the random vectors to be of different lengths in each source but the numeric vector must be of length equal to the number of data sources being used. Often, one wishes to generate random vectors of length equal to the length of standard vectors in each source. To do this most easily, issue a command such as: numobs.list<-ds.length('varname',type='split') where varname is an arbitrary vector of standard length in all sources. Then issue command: numobs<-unlist(numobs.list) to make numobs numeric rather than a list. Finally, declare samp.size=numobs as the first argument for the ds.rBinom function Please note that because (in this case) numobs is a clientside vector it should be specified without inverted commas (unlike the serverside vectors which may be used for the <size> and <prob> arguments [see below]).
size	a scalar that must be a positive integer - see "details" above. It cannot be specified as a numeric with a decimal point. You can also specify the <size> argument to be a serverside vector equal in length to the random number vector you want to generate and this allows the size to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.sizes) you must specify the argument as a character string (..., size="vector.of.sizes"...). If you simply wish to specify a single but different value in each source, then you can specify <size> as a scalar and use the <datasources> argument to create the random vectors one source at a time. Default value for <size> = 1 which simulates binary outcomes (all observations 0 or 1).
prob	a numeric scalar in range 0 > prob > 1 which specifies the probability of a positive response (i.e. 1 rather than 0) - see "details" above. Alternatively you can specify the <prob> argument to be a serverside vector equal in length to the random number vector you want to generate and this allows prob to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.probs) you must specify the argument as a character string (..., prob="vector.of.probs"...). If you simply wish to specify a single but different value in each source, then you can specify <prob> as a scalar and use the <datasources> argument to create the random vectors one source at a time. Default value for <prob> = 0.5 (equivalent to tossing an unbiased coin).
newobj	This a character string providing a name for the output random number vector which defaults to 'newObject' if no name is specified.
seed.as.integer	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.rBinom calls the serverside assign function setSeedDS to create the random seeds in each 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: eg if <seed.as.integer> = 32 and there are three studies, the ds.rBinom function will return: "$integer.seed.as.set.by.source", [1] 32 64 96, rather than the three vectors each of length 626 that represent the full seeds generated in each source. Default is FALSE.

`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)]

Details

An assign function that creates a vector of pseudorandom numbers in each data source. This function generates random numbers distributed with a binomial distribution - the situation that arises when individual observations can either be 0 or 1 with a given probability. The value observed for each observation (0 or 1) may formally be called a "Bernoulli trial". This being the case two arguments determine the binomial distribution to be generated using ds.rBinom: <size> and <prob>. The argument <prob> determines the probability of a positive response (i.e. the probability of observing 1 rather 0) in a single observation and it must lie strictly in the range 0 > prob >1. The argument <size> allows the observations to be considered in groups. For example if size=5 an observation consists of a group of 5 Bernoulli trials and the simulated outcome can either be 0,1,2,3,4 or 5. If <size> = 1 all observations are either 0 or 1 and this therefore allows one to simulate binary data. The argument <size> must be specified as an integer not as a numeric with a decimal point. The arguments of ds.rBinom also allow one to specify the length of the output vector in each source.

Value

Writes the pseudorandom number vector with the characteristics specified in the function call as a new serverside vector in each data source. Also returns key information to the clientside: the random seed trigger as specified by you in each source + (if requested) the full 626 length random seed vector this generated in each source (see info for the argument <return.full.seed.as.set>). The

ds.rBinom function also returns a vector reporting the length of the pseudorandom vector created in each source.

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.recodeLevels` *Recodes the levels of a factor vector*

Description

The function replaces the levels of a factor by the specified ones.

Usage

```
ds.recodeLevels(x = NULL, newCategories = NULL, newobj = NULL,
  datasources = NULL)
```

Arguments

`x,`	a character, the name of a factor variable.
`newCategories,`	a character vector, the new levels. Its length MUST be equal or greater to the current number of levels.
`newobj,`	a character, the name of the new factor vector. If no name is specified for the new variable it is named after the input variable with a suffixe '_new'.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources. By default an internal function looks for 'opal' objects in the environment and sets this parameter.

Details

It uses the R function 'levels()' on the client side to alter the current 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.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the variables
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # let s first check the levels in the categorical variable 'PM_BMI_CATEGORICAL'
  ds.levels(x='D$PM_BMI_CATEGORICAL')

  # Example1: merge the levels '2' and '3' to obtain only two levels (i.e. '1' and '2')
  # this is the same as recoding level '3' as '2' whilst keeping the same labels for the other
  # two levels.
  ds.recodeLevels(x='D$PM_BMI_CATEGORICAL', newCategories=c('1','2','2'), newobj='BMI_CAT_NEW1')
  ds.levels(x='BMI_CAT_NEW1')

  # Example2: add a 4th and empty level to categorical bmi to create a new variable
  # we know the current categories are '1', '2' and '3' so we add '4'
  ds.recodeLevels(x='D$PM_BMI_CATEGORICAL', newCategories=c('1','2','3','4'), newobj='BMI_CAT_NEW2')
  ds.levels(x='BMI_CAT_NEW2')

  # Example3: re-label the levels of the categorical bmi "low", "mid" and "high"
  ds.recodeLevels(x='D$PM_BMI_CATEGORICAL', newCategories=c('low','mid','high'),
                  newobj='BMI_CAT_NEW3')
  ds.levels(x='BMI_CAT_NEW3')

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


## End(Not run)
```

`ds.recodeValues` *ds.recodeValues calling recodeValuesDS1 and recodeValuesDS2*

Description

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, force.output.format = "no",
  newobj = NULL, datasources = NULL, notify.of.progress = FALSE)
```

Arguments

var.name	a character string providing the name for the vector representing the variable to be recoded
values2replace.vector	a numeric or character vector specifying the values in the vector specified by the argument <var.name> that are to be replaced by new values as specified in the new.values.vector. Example 1, with the two arguments values2replace.vector=c(0,1,2) and new.values.vector=c(20,27.5,37): 0s in the declared <var.name> will be replaced by the value 20, 1s by 27.5 and 2s by 37. If there are any values in the <var.name> vector other than 0, 1 or 2 they will remain unchanged. Example 2, with the two arguments values2replace.vector=c(0,1,2,NA) and new.values.vector=c("Norm","Overwt", "Obese",999): 0s in the declared <var.name> will be replaced by the character value "Norm", 1s by "Overwt", 2s by "Obese" and NAs by 999. As context to these two examples, these represent the recoding of a grouped BMI variable (taking values 0,1,2 and NA) with the numeric value representing the approximate mean of each group (example 1) and category names and an explicit value for missing (example 2). Each value in <values2replace.vector> can only appear once and the length of <values2replace.vector> must be equal to the length of <new.values.vector>
new.values.vector	a numeric or character vector specifying the new values to which the specified values in the vector <var.name> are to be changed. The length of <new.values.vector> must be equal to the length of <values2replace.vector> but more than one value in the latter can be changed to the same value in the former - so <new.values.vector> can include repeated values
force.output.format	character string. If this is 'numeric' the recoded (output) vector will be numeric - any non-numeric values in the <new.values.vector> will appear as NaN in the recoded (output) vector. If the <force.output.format> argument is declared as 'character' all values in the recoded output vector will be in character format. The <force.output.format> argument defaults to "no" and in that case, if the vector identified by the <values2replace.vector> argument is itself numeric and if all values in the <new.values.vector> are numeric, the recoded output vector will also be numeric. Otherwise, it will be coerced to character format.
newobj	This a character string providing a name for the recoded vector representing the primary output of the ds.recodeValues() function. This defaults to '<var.name>_recoded' if no name is specified where <var.name> is the first argument of ds.recodeValues()
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

Recodes individual values with new individual values. This can apply to numeric values, character values and NAs. One particular use of ds.recodeValues is to convert NAs to an explicit value or vice-versa. Please see ds.Boole to recode a RANGE of values with a new value. Please note that if you wish to do no more than replace NAs with a new code (e.g. 999) there is a restriction imposed by the fact that if the <values2replace> argument is specified as c(NA) or NA and the <new.values.vector> argument as c(999) the function will fail because it will not properly interpret the first (all values missing) argument as a valid scalar or vector of length 1. To work around this restriction please specify the new <values2replace> argument as c(x,NA) and the <new.values.vector> argument as c(x,999) where x is one of the other valid (non-missing) levels in the vector to be recoded. This will leave the x values unchanged but because the <values2replace> argument is not all missing it will correctly recognise that there are two values to change one of which happens to be NA to be replaced by 999 and the other will be 'replaced' by its pre-existing value.

Value

the object specified by the <newobj> argument (or default name '<var.name>_recoded'). 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 - the function returns a range of possible studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message("newobj") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("newobj") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

DataSHIELD Development Team

---

ds.replaceNA *Replaces the missing values in a 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, the name of the vector to process.
forNA	a list which contains the replacement value(s), a vector one or more values for each study. The length of the list must be equal to the number of servers the analyst is connected to.
newobj	a character, the name of the new vector in which missing values have been replaced. If no name is specified the default name is the name of the original vector followed by the suffix '.noNA' e.g. 'LAB_HDL.noNA' if the name of the vector is 'LAB_HDL'.
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

This function is used when the analyst prefer 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 hold hold both the vector with and without missing values. The latter is, by default, given a different that indicates its 'completeness'.

Value

a new vector or table structure with the same class is stored on the server site.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign all the stored variables.
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # Replace missing values in variable 'LAB_HDL' by the mean value in each study
  # first let us get the mean value for 'LAB_HDL' in each study
  ds.mean(x='D$LAB_HDL', type='split')

  # replace missing values in the variable 'LAB_HDL' in dataf frame 'D' by
  # the mean value and name the new variable 'HDL.noNA'.
  ds.replaceNA(x='D$LAB_HDL', forNA=list(1.569416, 1.556648), newobj='HDL.noNA')
```

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


## End(Not run)
```

---

ds.reShape *ds.reShape calling assign function reShapeDS*

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,	the name of the data frame to be reshaped. The user must set the name as a character string in inverted commas. For example: data.name='data.frame.name'
varying,	names of sets of variables in the wide format that correspond to single variables in long format (typically what may be called 'time-varying' or 'time-dependent' variables). For example, varying=c('outcome1.1', 'outcome1.2', 'outcome1.3', 'outcome1.4', 'outcome1.5', 'outcome1.6')
v.names,	the names of variables in the long format that correspond to multiple variables in the wide format - for example, the single vector 'sbp' in long format may reflect 'sbp7', 'sbp11', 'sbp15' in wide format (measured systolic blood pressure at ages 7, 11 and 15 years). In the long format these simply represent 3 different records with the systolic bp recorded in one column (i.e. sbp) and age of measurement recorded in another column (e.g. age). For example, v.names=c('outcome1','outcome2') specifies that 'outcome1' and 'outcome2' in long format vary with time and may generate multiple columns in 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. In the example, given under param v.names, it is the 'age' variable that discriminates the time at which each measurement was taken. For example, timevar.name='time.name'
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. For example, if there is a numeric ID, all observations for the individual with ID 23 may have the value 23 in an 'individual.ID' vector which can be declared as <idvar.name>: idvar.name='individual.ID'

drop,	a vector of names of variables to drop before reshaping. This can simplify the resultant output. For example, drop=c('bmi.26','pm10.16','survtime','censor')
direction,	a character string, 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. For example, in long format if sep='.', and systolic blood pressure is held in a column 'sbp', and age is recorded in years in a vector 'age' then depending how you set things up the column for sbp at age 7 in the wide format may be called 'sbp.7' or 'sbp.age.7'
newobj	A character string specifying the name of the vector to which the output vector is to be written. If no <newobj> argument is specified, the output vector defaults to 'newObject'.
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)]

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

Value

a reshaped data.frame converted from long to wide format or from wide to long format which is written to the serverside and given the name provided as the <newobj> argument or 'newObject' if no name is specified. In addition, two validity messages are returned to the clientside 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.dataFrame() also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you can use the ds.message function. If you type ds.message('newobj') it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message('newobj') will return the message: 'ALL OK: there are no studysideMessage(s) on this datasource'

Author(s)

Demetris Avraam, Paul Burton for DataSHIELD Development Team

ds.rm *ds.rm calling aggregate function rmDS*

Description

deletes an R object on the serverside

Usage

```
ds.rm(x.name = NULL, datasources = NULL)
```

Arguments

x.name,	the name of the object to be deleted specified in inverted commas. For example: x.name='object.name'.
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)]

Details

this clientside function calls a serverside function based on the rm() function in native R. This is the aggregate function rmDS. The fact that it is an aggregate function maybe surprising because it modifies an object on the serverside, 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.

Value

the specified object is deleted from the serverside. If this is successful the message "Object <x.name> successfully deleted" is returned to the clientside (where x.name is the name of the object to be deleted). If the object to be deleted is already absent on a given source, that source will return the message: "Object to be deleted, i.e. <x.name> , does not exist so does not need deleting". Finally, if the specified name of the object to be deleted is too long (>nfilter.stringShort) there is a potential disclosure risk (active code hidden in the name) and the serverside function returns a message such as: "Disclosure risk, number of characters in x.name must not exceed nfilter.stringShort which is currently set at: 25" where '25' is the current setting of the R_Option value of nfilter.stringShort.

Author(s)

Paul Burton for DataSHIELD Development Team

ds.rNorm *ds.rNorm calling rNormDS and setSeedDS*

Description

Generates random (pseudorandom) numbers with a normal distribution

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	the length of the random number vector to be created in each source. <samp.size> can be a numeric scalar and this then specifies the length of the random vectors in each source to be the same. If it is a numeric vector it enables the random vectors to be of different lengths in each source but the numeric vector must be of length equal to the number of data sources being used. Often, one wishes to generate random vectors of length equal to the length of standard vectors in each source. To do this most easily, issue a command such as: numobs.list<-ds.length('varname',type='split') where varname is an arbitrary vector of standard length in all sources. Then issue command: numobs<-unlist(numobs.list) to make numobs numeric rather than a list. Finally, declare samp.size=numobs as the first argument for the ds.rNorm function Please note that because (in this case) numobs is a clientside vector it should be specified without inverted commas (unlike the serverside vectors which may be used for the <mean> and <sd> arguments [see below]).
mean	a numeric scalar specifying the mean of the generating normal distribution in each data source. Alternatively you can specify the <mean> argument to be a serverside vector equal in length to the random number vector you want to generate and this allows the mean to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.means) you must specify the argument as a character string (..., mean="vector.of.means"...). If you simply wish to specify a single but different value in each source, then you can specify <mean> as a scalar and use the <datasources> argument to create the random vectors one source at a time. Default value for <mean> = 0.
sd	a numeric scalar specifying the standard deviation of the generating normal distribution in each data source. Alternatively you can specify the <sd> argument to be a serverside vector equal in length to the random number vector you want to generate and this allows the sd to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.sds) you must specify the argument as a character string (..., sd="vector.of.sds"...) If you simply wish to specify a single but different value in each source, then you can use the <datasources> argument to create the random vectors one source at a time. Default value = 1.

`newobj` This a character string providing a name for the output random number vector which defaults to 'newObject' if no name is specified.

`seed.as.integer`
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.rNorm calls the serverside assign function setSeedDS to create the random seeds in each 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: eg if <seed.as.integer> = 32 and there are three studies, the ds.rNorm function will return: "$integer.seed.as.set.by.source", [1] 32 64 96, rather than the three vectors each of length 626 that represent the full seeds generated in each source. Default is FALSE.

`force.output.to.k.decimal.places`
scalar integer. Forces the output random number vector to have k decimal places. If 0 rounds it coerces decimal random number output to integer, a k in range 1-8 forces output to have k decimal places. If k = 9, no rounding occurs of native output. Default=9.

`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)]

Details

An assign function that creates a vector of pseudorandom numbers in each data source. This function generates normally distributed random numbers. The function's arguments specify the length of the output vector in each source and the mean and standard deviation (sd) of the normal distribution to generate from.

Value

Writes the pseudorandom number vector with the characteristics specified in the function call as a new serverside vector in each data source. Also returns key information to the clientside: the random seed trigger as specified by you in each source + (if requested) the full 626 length random seed vector this generated in each source (see info for the argument <return.full.seed.as.set>). The ds.rNorm function also returns a vector reporting the length of the pseudorandom vector created in each source.

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.rowColCalc` *Computes sums and means of rows or columns of numeric matrix or data frame*

Description

The function is similar to R base functions 'rowSums', 'colSums', 'rowMeans' and 'colMeans' with some restrictions.

Usage

```
ds.rowColCalc(x = NULL, operation = NULL, newobj = NULL,
  datasources = NULL)
```

Arguments

`x`	a character, the name of a matrix or a dataframe
`operation`	a character string which indicates the operation to carry out: "rowSums", "colSums", "rowMeans" or "colMeans".
`newobj`	the name of the new object. If this argument is set to NULL, the name of the new variable, set by default, is 'rowColCalc_out'.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

The results of 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.

Value

nothing is returned to the client, the new object is stored on the server side.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign two variables
  myvar  <-  list("LAB_TSC","LAB_HDL")
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # calculate the sum of each row of the above assigned dataset (default name 'D')
  ds.rowColCalc(x='D', operation='rowSums', newobj='rsum_D')

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


## End(Not run)
```

---

ds.rPois *ds.rPois calling rPoisDS and setSeedDS*

Description

Generates random (pseudorandom) numbers (non-negative integers) with a Poisson distribution

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	the length of the random number vector to be created in each source. <samp.size> can be a numeric scalar and this then specifies the length of the random vectors in each source to be the same. If it is a numeric vector it enables the random vectors to be of different lengths in each source but the numeric vector must be of length equal to the number of data sources being used. Often, one wishes to generate random vectors of length equal to the length of standard vectors in each source. To do this most easily, issue a command such as: numobs.list<-ds.length('varname',type='split') where varname is an arbitrary vector of standard length in all sources. Then issue command: numobs<-unlist(numobs.list)

to make numobs numeric rather than a list. Finally, declare samp.size=numobs as the first argument for the ds.rPois function Please note that because (in this case) numobs is a clientside vector it should be specified without inverted commas (unlike the serverside vectors which may be used for the <lambda> argument [see below]).

`lambda` a numeric scalar specifying the expected count of the Poisson distribution used to generate the random counts. If you wish to specify a different value in each source, then you can use the <datasources> argument to create the random vectors one source at a time, changing lambda as required. Alternatively you can specify the <lambda> argument to be a serverside vector equal in length to the random number vector you want to generate and this allows lambda to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.lambdas) you must specify the argument as a character string (..., lambda="vector.of.lambdas"...). If you simply wish to specify a single but different value in each source, then you can specify <lambda> as a scalar and use the <datasources> argument to create the random vectors one source at a time. Default value for <lambda> = 1.

`newobj` This a character string providing a name for the output random number vector which defaults to 'newObject' if no name is specified.

`seed.as.integer`
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.rPois calls the serverside assign function setSeedDS to create the random seeds in each 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: eg if <seed.as.integer> = 32 and there are three studies, the ds.rPois function will return: "$integer.seed.as.set.by.source", [1] 32 64 96, rather than the three vectors each of length 626 that represent the full seeds generated in each source. Default is FALSE.

`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)]

Details

An assign function that creates a vector of pseudorandom numbers in each data source. This function generates random numbers distributed with a Poisson distribution - non-negative integer values with an expected count fully specified by a single argument lambda. In addition, the ds.rPois function's arguments specify the length of the output vector in each source.

Value

Writes the pseudorandom number vector with the characteristics specified in the function call as a new serverside vector in each data source. Also returns key information to the clientside: the random seed trigger as specified by you in each source + (if requested) the full 626 length random seed vector this generated in each source (see info for the argument <return.full.seed.as.set>). The ds.rPois function also returns a vector reporting the length of the pseudorandom vector created in each source.

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.rUnif` *ds.rUnif calling rUnifDS and setSeedDS*

Description

Generates random (pseudorandom) numbers with a uniform distribution

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` the length of the random number vector to be created in each source. <samp.size> can be a numeric scalar and this then specifies the length of the random vectors in each source to be the same. If it is a numeric vector it enables the random vectors to be of different lengths in each source but the numeric vector must be of length equal to the number of data sources being used. Often, one wishes to generate random vectors of length equal to the length of standard vectors in each source. To do this most easily, issue a command such as: numobs.list<-ds.length('varname',type='split') where varname is an arbitrary vector of standard length in all sources. Then issue command: numobs<-unlist(numobs.list) to make numobs numeric rather than a list. Finally, declare samp.size=numobs

as the first argument for the ds.rUnif function Please note that because (in this case) numobs is a clientside vector it should be specified without inverted commas (unlike the serverside vectors which may be used for the <min> and <max> arguments [see below]).

`min` a numeric scalar specifying the minimum of the range across which the random numbers will be generated in each source. Alternatively you can specify the <min> argument to be a serverside vector equal in length to the random number vector you want to generate. This allows min to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.mins) you must specify the argument as a character string (..., min="vector.of.mins"...). If you simply wish to specify a single but different value in each source, then you can specify <min> as a scalar and use the <datasources> argument to create the random vectors one source at a time. Default value for <min> = 0.

`max` a numeric scalar specifying the maximum of the range across which the random numbers will be generated in each source. Alternatively you can specify the <max> argument to be a serverside vector equal in length to the random number vector you want to generate. This allows max to vary by observation in the dataset. If you wish to specify a serverside vector in this way (e.g. called vector.of.maxs) you must specify the argument as a character string (..., max="vector.of.maxs"...). If you simply wish to specify a single but different value in each source, then you can specify <max> as a scalar and use the <datasources> argument to create the random vectors one source at a time. Default value for <max> = 1

`newobj` This a character string providing a name for the output random number vector which defaults to 'newObject' if no name is specified.

`seed.as.integer`

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 calls the serverside assign function setSeedDS to create the random seeds in each 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: eg if <seed.as.integer> = 32 and there are three studies, the ds.rUnif function will return: "$integer.seed.as.set.by.source", [1] 32 64 96,

rather than the three vectors each of length 626 that represent the full seeds generated in each source. Default is FALSE.

`force.output.to.k.decimal.places`
: scalar integer. Forces the output random number vector to have k decimal places. If 0 rounds it coerces decimal random number output to integer, a k in range 1-8 forces output to have k decimal places. If k = 9, no rounding occurs of native output. NOTE IF YOU WANT CATEGORIES WITH EQUAL PROBABILITY (PARTICULARLY WHEN CREATING INTEGERS) YOU SHOULD EXTEND THE SIMULATION RANGE AT BOTH ENDS: IF K = 0 AND YOU WISH TO GENERATE INTEGERS WITH EQUAL PROBABILITY IN THE RANGE 1-10, YOU SHOULD SPECIFY <min>=0.5 AND <max>=10.5. Default value for k =9.

`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)]

Details

An assign function that creates a vector of pseudorandom numbers in each data source. This function generates random numbers distributed with a uniform probability across a range specified with a minimum and maximum. The function's arguments specify the length of the output vector in each source and the minimum and maximum of the range across which the uniform distribution to be generated.

Value

Writes the pseudorandom number vector with the characteristics specified in the function call as a new serverside vector in each data source. Also returns key information to the clientside: the random seed trigger as specified by you in each source + (if requested) the full 626 length random seed vector this generated in each source (see info for the argument <return.full.seed.as.set>). The ds.rUnif function also returns a vector reporting the length of the pseudorandom vector created in each source.

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.scatterPlot` *Generates non-disclosive scatter plots*

Description

This function uses two disclosure control methods to generate non-disclosive scatter plots of two continuous variables

Usage

```
ds.scatterPlot(x = NULL, y = NULL, method = "deterministic", k = 3,
  noise = 0.25, type = "split", datasources = NULL)
```

Arguments

x	a character, the name of a numeric vector, the x-variable.
y	a character, the name of a numeric vector, the y-variable.
method	a character which specifies the method that is used to generated non-disclosive coordinates to be displayed in a scatter plot. If the method is set to 'deteministic' (default), then the scatter plot shows the scaled centroids of each k nearest neighbours of the original variables where the value of k is set by the user. If the method is set to 'probabilistic', then the scatter plot shows the original data 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 each variable. This percentage is specified by the user in the argument noise.
k	the number of the nearest neghbours for which their centroid is calculated. 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'.
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'. Any value of noise is ignored if the argument method is set to 'deterministic'. The user can choose any value for noise equal to or greater than the pre-specified threshold 'nfilter.noise'.
type	a character which represents the type of graph to display. A scatter plot for combined data is generated when the type is set to 'combine'. One scatter plot for each single study is generated when the type is set to 'split' (default).
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

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 neigbours 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.The value of k in this deterministic approach, 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. 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 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. 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.

Value

one or more scatter plots depending on the argument `type`

Author(s)

Demetris Avraam for DataSHIELD Development Team

Examples

```
## Not run:

  # load the file that contains the login details
  data(logindata)

  # login to the servers
  opals <- opal::datashield.login(logins=logindata, assign=TRUE)

  # Example 1: generate a scatter plot for each study separately (the default behaviour)
  ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', type="split")

  # Example 2: generate a combined scatter plot with the default deterministic method
  ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', k=3,
                  method='deterministic')

  # Example 3: if a variable is of type factor the scatter plot is not created
  ds.scatterPlot(x='LD$PM_BMI_CATEGORICAL', y='LD$LAB_GLUC_ADJUSTED')

  # Example 4: same as Example 2 but with k=50
  ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', k=50,
                  method='deterministic', type='combine')

 # Example 5: same as Example 2 but with k=1740 (here we see that as k increases we have big
              utility loss)
 ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', k=1740,
                  method='deterministic', type='combine')
```

```
 # Example 6: same as Example 5 but for split analysis
 ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', k=1740,
                 method='deterministic', type='split')

# Example 7: if k is less than the specified threshold then the scatter plot is not created
 ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', k=2,
                 method='deterministic')

 # Example 8: generate a combined scatter plot with the probabilistic method
ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', method='probabilistic',
                 type='combine')

# Example 9: generate a scatter plot with the probabilistic method for each study separately
ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', method='probabilistic',
                 type='split')

 # Example 10: same as Example 9 but with higher level of noise
ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', method='probabilistic',
                 noise=0.5, type='split')

# Example 11: if 'noise' is less than the specified threshold then the scatter plot is not created
ds.scatterPlot(x='LD$PM_BMI_CONTINUOUS', y='LD$LAB_GLUC_ADJUSTED', method='probabilistic',
                 noise=0.1, type='split')

 # clear the Datashield R sessions and logout
 opal::datashield.logout(opals)


## End(Not run)
```

---

ds.seq *ds.seq calling seqDS*

Description

ds.seq calling assign function seqDS

Usage

```
ds.seq(FROM.value.char = "1", BY.value.char = "1",
  LENGTH.OUT.value.char = NULL, ALONG.WITH.name = NULL,
  newobj = "newObj", datasources = NULL)
```

Arguments

FROM.value.char
: the starting value for the sequence expressed as an integer in character form. e.g. FROM.value.char="1" will start at 1, FROM.value.char="-10" will start at -10. Default = "1"

`BY.value.char` the value to increment each step in the sequence expressed as a numeric e.g. BY.value.char="10" will increment by 10, while BY.value.char="-3.37" will reduce the value of each sequence element by -3.37. Default = "1" but does not have to be integer

`LENGTH.OUT.value.char`
The length of the sequence at which point its extension should be stopped. e.g. LENGTH.OUT.value.char="1000" will generate a sequence of length 1000. Default = NULL (must be specified) but must be a positive integer

`ALONG.WITH.name`
For convenience, rather than specifying a value for LENGTH.OUT it can often be better to specify a variable name as the <ALONG.WITH.name> argument. e.g. ALONG.WITH.name = "vector.name". This can be particularly useful in DataSHIELD where the length of the sequence you need to generate in each data set depends on the standard length of vectors in that data set and this will in general vary.

`newobj` This a character string providing a name for the output sequence vector which defaults to 'newObj' 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)].

Details

Calls an assign function that uses the native R function seq() to create any one of a flexible range of sequence vectors that can then be used to help manage and analyse data. As it is an assign function the resultant vector is written as a new object onto all of the specified data source servers. For the purposes of creating the DataSHIELD equivalent to seq() in native R we have used all of the original arguments (see below) except the <to> argument. This simplifies the function and prevents some combinations of arguments that lead to an error in native R. The effect of the <to> argument - see help(seq) in native R - is to specify the terminal value of the sequence. However, when using seq() one can usually specify other arguments (see below) to mimic the desire effect of <to>. These include: <from>, the starting value of the sequence; <by>, its increment (+ or -), and <length.out> the length of the final vector in each data source.

Value

the object specified by the <newobj> argument (or default name newObj) which is written to the serverside. As well as writing the output object as <newobj> on the serverside, 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.seq() also returns any studysideMessages that can explain the error in creating the full output object. As well as appearing on the screen at run time,if you wish to see the relevant studysideMessages at a later date you

can use the ds.message function. If you type ds.message("<newobj>") it will print out the relevant studysideMessage from any datasource in which there was an error in creating <newobj> and a studysideMessage was saved. If there was no error and <newobj> was created without problems no studysideMessage will have been saved and ds.message("<newobj>") will return the message: "ALL OK: there are no studysideMessage(s) on this datasource".

Author(s)

Paul Burton for DataSHIELD Development Team

`ds.setSeed` *ds.setSeed calling setSeedDS*

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 scalar or a NULL which primes the random seed in each data source. 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 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. Thus, by default, when you are generating pseudorandom number vectors in a series of different data sources using the standard DataSHIELD functions the vectors will be different in each source. Given the inbuilt choice of arguments for set.seed() that are fixed in DataSHIELD's setSeedDS function, if a given priming integer is specified as the argument <seed.as.integer> in ds.setSeed the .Random.seed vector that will be generated will be the same on any data source internationally (regardless of the flavour of R). Please note that the first two elements of .Random.seed do not vary meaningfully, in particular,

before a seed is set, element 2 varies between different R platforms, but once the seed has been set, it becomes 624 which happens then to be the length of the remaining 624 elements (3-626) of .Random.seed which provide the meaningful component of the random number seed.

`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)]

Details

ds.setSeed calls the aggregate function setSeedDS in each data source, passing it a single integer value which acts as a trigger value in that source to generate an instance of the full pseudorandom number seed that is a vector of integers of length 626 called .Random.seed. Each time a new pseudorandom number is generated, the current .Random.seed vector provides a deterministic but very close to behaviourally random way to generate that pseudorandom number and to completely regenerate the random seed vector. Unusually, because setSeedDS is effectively the same as the set.seed() function in native R, although it writes a new object to the serverside (i.e. the integer vector of length 626 known as .Random.seed [see info for the argument <seed.as.integer>]) because this is done directly via calling a native R function this has been set up as an aggregate function not an assign function.

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. 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 (626 elements) that is .Random.seed itself.

Author(s)

Paul Burton for DataSHIELD Development Team

---

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 oprator and a threshold. The object to subset from must be a vector (factor, numeric or charcater) 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 opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

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 refered 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")
  opals <- 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(opals)


## 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 opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

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')
  opals <- 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(opals)


## End(Not run)
```

`ds.summary` *Generates the summary of an object*

Description

Provides some insight about an object. Unlike the similar R function only a limited class of objects can be used as input to reduce the risk of disclosure.

Usage

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

Arguments

`x`	a numeric or factor variable
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

The class and size of the object are returned and various other information are also returned depending of the class of the object. Potentially disclosive information such as the minimum and maximum values of numeric vectors are not returned. The summary is given for each study separately.

Value

the returned information depends on the class of the objects.

Author(s)

Gaye, A.

Examples

```
## Not run:

  # load the login data
  data(logindata)

  # login and assign all the variable held in the opal database
  opals <- datashield.login(logins=logindata,assign=TRUE)

  # Example 1: suummary of a numerical variable
  ds.summary(x='D$LAB_TSC')

  # Example 1: suummary of a binary variable
  ds.summary(x='D$GENDER')

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


## End(Not run)
```

---

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 ouput: 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 (deafult) 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 opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

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 opal) 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
opals  <-  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(opals)


## 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 ouput: 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 (deafult) 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 opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

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 opal 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
opals  <-  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(opals)


## End(Not run)
```

ds.tapply *ds.tapply calling tapplyDS*

Description

Apply one of a selected range of functions to summarize an outcome variable over one or more indexing factors and write the resultant summary to the clientside

Usage

```
ds.tapply(X.name = NULL, INDEX.names = NULL, FUN.name = NULL,
  datasources = NULL)
```

Arguments

X.name,	the name of the variable to be summarized. The user must set the name as a character string in inverted commas. For example: X.name="var.name"
INDEX.names,	the name of a single factor or a vector of names of factors to index the variable to be summarized. Each name must be specified in inverted commas. For example: INDEX.names="factor.name" or INDEX.names=c("factor1.name", "factor2.name", "factor3.name"). The native R tapply function can coerce non-factor vectors into factors. However, this does not always work when using the DataSHIELD ds.tapply/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
FUN.name,	the name of one of the allowable summarizing functions to be applied specified in inverted commas. The present version of the function allows the user to choose one of five summarizing functions. These are "N" (or "length"), "mean","sd", "sum", or "quantile". For more information see Details.
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)]

Details

A clientside function calling an aggregate serverside function that uses the native R function tapply() to apply 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 native R tapply function is very flexible and the range of allowable summarizing functions is much more restrictive for the DataSHIELD ds.tapply function. This is to protect against disclosure risk. At present the allowable functions are: N/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). Should other functions be required in the future then, provided they are non-disclosive, the DataSHIELD development team could work on them if requested. As an aggregate function ds.tapply returns the summarized values to the clientside. In order to protect against disclosure the number of observations in each summarizing group in each source is calculated and if any of these fall 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 wish 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 clientside. Rather, it can be written as a vector to the serverside at each source (which, like any other serverside object, cannot then be seen, abstracted or copied). This can be done using ds.tapply.assign which calls an assign function tapplyDS.assign which acts just like the aggregate function tapplyDS but it writes the results as a newobj to the serverside and does not test the number of observations in each group against nfilter.tab. 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 serverside functions tapplyDS and tapplyDS.assign both start 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.

Value

an array of the summarized values created by the tapplyDS function. This array is returned to the clientside. It has the same number of dimensions as INDEX.

Author(s)

Paul Burton, Demetris Avraam for DataSHIELD Development Team

`ds.tapply.assign` *ds.tapply.assign calling tapplyDS.assign*

Description

Apply 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 serverside

Usage

```
ds.tapply.assign(X.name = NULL, INDEX.names = NULL, FUN.name = NULL,
  newobj = "tapply.out", datasources = NULL)
```

Arguments

X.name,	the name of the variable to be summarized. The user must set the name as a character string in inverted commas. For example: X.name="var.name"
INDEX.names,	the name of a single factor or a vector of names of factors to index the variable to be summarized. Each name must be specified in inverted commas. For example: INDEX.names="factor.name" or INDEX.names=c("factor1.name", "factor2.name", "factor3.name"). The native R tapply function can coerce non-factor vectors into factors. However, this does not always work when using the DataSHIELD ds.tapply/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
FUN.name,	the name of one of the allowable summarizing functions to be applied specified in inverted commas. The present version of the function allows the user to choose one of five summarizing functions. These are "N" (or "length"), "mean","sd", "sum", or "quantile". For more information see Details.
newobj	A character string specifying the name of the vector to which the output vector is to be written. If no <newobj> argument is specified, the output vector defaults to "tapply.out".
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)]

Details

A clientside function calling an assign serverside function that uses the native R function tapply() to apply 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 native R tapply function is very flexible and the range of allowable summarizing functions is much more restrictive for the DataSHIELD ds.tapply function. This is to protect against disclosure risk. At present 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). Should other functions be required in the future then, provided they are non-disclosive, the DataSHIELD development team could work on them if requested. As an assign function tapplyDS.assign writes the summarized values to the serverside. Because unlike the aggregate function tapplyDS, tapply.assign returns no results to the clientside, it is fundamentally non-disclosive and the number of observations in each unique indexing group does not need to be evaluated against nfilter.tab (the minimum allowable non-zero count in a contingency table). This means that tapplyDS.assign can be used, 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. This will almost inevitably generate some indexing groups smaller than nfilter.tab but as the results are simply written as newobj to the serverside rather than returned to the clientside there is

no overt disclosure risk. 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 serverside functions tapplyDS and tapplyDS.assign both start 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.

Value

an array of the summarized values created by the tapplyDS.assign function. This array is written as a newobj onto the serverside. It has the same number of dimensions as INDEX.

Author(s)

Paul Burton, Demetris Avraam for DataSHIELD Development Team

---

ds.testObjExists *Checking that a correct version of a data object exists on a data source server*

Description

This function checks that a specified data object exists or has been correctly created on a specified set of data servers (may be only one data server).

Usage

```
ds.testObjExists(test.obj.name = NULL, datasources = NULL)
```

Arguments

`test.obj.name`	a character string specifying the name of the object to search for e.g. "TID.f"
`datasources`	specifies the particular opal object(s) to use, if it is not specified the default set of opals will be used. The default opals are always called default.opals. This parameter is set without inverted commas: e.g. datasources=opals.em or datasources=default.opals If you wish to specify the second opal server in a set of three, the parameter is specified: e.g. datasources=opals.em[2]. If you wish to specify the first and third opal servers in a set specify: e.g. datasources=opals.em[2,3]

Details

The data object to search for is defined as a character string to the argument test.obj.name. The set of data servers to search on is provided as the argument datasources. 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, ds.asFactor

Value

If the specified data object exists and is of a valid class (i.e. is not null) in every one of the datasources specified by the datasources argument, the function returns a list with a single element $return.message which is of the form: "A valid copy of data object <TID.f> exists in all specified data sources". If the specified object is non-existant in at least one of the specified data sources or it exists but is of class "null", the $return message is of the form: "Error: A valid data object TID.h does NOT exist in ALL specified data sources" "It is either ABSENT and/or has no valid content/class, for details see return.info above". The list return.info then includes a list of each data source with an indication of whether the data object being tested exists at all e.g. for study 3 return of the list object $return.info$study3$test.obj.exists = FALSE implies that the tested object does not exist even as a name in study 3 while return of the list object $return.info$study3$test.obj.class = "ABSENT" implies that even if the tested object exists as a name in study 3 it does not have a valid class (so contains no data etc)

Author(s)

Burton PR

`ds.unList` *ds.unList calling aggregate function unListDS*

Description

this function is based on the native R function unlist which 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, recursive = TRUE, newobj = NULL,
  datasources = NULL)
```

Arguments

`x.name`	the name of the input object to be unlisted. It must be specified in inverted commas e.g. x.name="input.object.name"
`recursive`	logical, if FALSE the function will not recurse beyond the first level items in x (e.g. the N data sources in many DataSHIELD settings. Default = TRUE so recursion includes all levels.
`newobj`	the name of the new output variable. If this argument is set to NULL, the name of the new variable is defaulted to <x.name>.unlist
`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 an explicit <datasources> argument is to be set, it should be specified 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)]

Details

See details of the native R function unlist. Unlike most other class coercing functions the serverside function that is called is an aggregate function rather than an assign function. This is because the datashield.assign function in opal deals specially with a created object (<newobj>) if it is of class list. Reconfiguring the function as an aggregate function works around this problem. 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 a list 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 eg '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

Value

the object specified by the <newobj> argument (or by default <x.name>.unlist if the <newobj> argument is NULL) which is written to the serverside. In addition, two validity messages are returned. The first confirms an output object has been created, the second states its class. The way that as.list coerces objects to list depends on the class of the object, and so the class of the unlisted output will depend on how the original list was formed - see details

Author(s)

Amadou Gaye, Paul Burton, for DataSHIELD Development Team

`ds.var` *ds.var calling aggregate function varDS*

Description

Computes the variance of a given vector This function is similar to the R function `var`.

Usage

```
ds.var(x = NULL, type = "split", checks = FALSE,
  datasources = NULL)
```

Arguments

`x`	a character, the name of a numerical vector.
`type`	a character which represents the type of analysis to carry out. If `type` is set to 'combine', 'combined', 'combines' or 'c', a global variance is calculated if `type` is set to 'split', 'splits' or 's', the variance is calculated separately for each study. if `type` is set to 'both' or 'b', both sets of outputs are produced

checks	a Boolean indicator of whether to undertake optional checks of model components. Defaults to checks=FALSE to save time. It is suggested that checks should only be undertaken once the function call has failed
datasources	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as dataframe, from opal datasources.

Details

It is a wrapper for the server side function. The server side function returns a list with the sum of the input variable, the sum of squares of the input variable, the number of missing values, the number of valid values, the number of total lenght of the variable, and a study message indicating whether the number of valid is less than the disclosure threshold. The variance is calculated at the client side by the formula $

$$var(X)$$

$

Value

a list including: Variance.by.Study = estimated variance in each study separately (if type = split or both), with Nmissing (number of missing observations), Nvalid (number of valid observations), Ntotal (sum of missing and valid observations) also reported separately for each study; Global.Variance = Variance, Nmissing, Nvalid, Ntotal across all studies combined (if type = combine or both); Nstudies = number of studies being analysed; ValidityMessage indicates whether a full analysis was possible or whether one or more studies had fewer valid observations than the nfilter threshold for the minimum cell size in a contingency table.

Author(s)

Amadou Gaye, Demetris Avraam, for DataSHIELD Development Team

Examples

```
## Not run:

  # load that contains the login details
  data(logindata)

  # login and assign specific variable(s)
  myvar <- list('LAB_TSC')
  opals <- datashield.login(logins=logindata,assign=TRUE,variables=myvar)

  # Example 1: compute the pooled variance of the variable 'LAB_TSC' - default behaviour
  ds.var(x='D$LAB_TSC')

  # Example 2: compute the variance of each study separately
  ds.var(x='D$LAB_TSC', type='split')

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

```
## 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 = "math_output",
  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_output'.
`datasources`	a list of opal object(s) obtained after login in to opal servers; these objects hold also the data assign to R, as `dataframe`, from opal datasources.

Details

In DataSHIELD it is possible to perform an operation on vectors by just using the relevant R symbols (e.g. '+' for addtion, '*' for multiplication, '-' for substraction 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 substraction 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')
  opals <- 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(opals)


## End(Not run)
```

Index