Package 'ForecastFramework'

January 16, 2020

Title A Basis for Modular Model Creation

Version 0.10.3

Maintainer Joshua Kaminsky <jkaminsky@jhu.edu>

Description Create modular models. Quickly prototype models whose input includes (multiple) time series data. Create pieces of model use cases separately, and swap out particular models as desired. Create modeling competitions, data processing pipelines, and re-useable models.

Depends R6, R (>= 2.10.0)

Imports abind,lubridate,dplyr,reshape2,magrittr,tibble

Suggests testthat,knitr,rmarkdown

Enhances surveillance

Encoding UTF-8

License GPL-3

LazyData true

RoxygenNote 6.1.99.9001

Collate 'AbstractClasses.R' 'DataContainers.R' 'Forecasts.R' 'SimulatedIncidenceMatrix.R' 'IncidenceForecast.R' 'IncidenceMatrix.R' 'Models.R' 'SimpleForecast.R' 'MoveAheadModel.R' 'ObservationList.R'

VignetteBuilder knitr

NeedsCompilation no

Author Joshua Kaminsky [aut, cre],
Justin Lessler [aut],
Nicholas Reich [aut]

Repository CRAN

Date/Publication 2020-01-16 15:30:02 UTC

R topics documented:

`AbstractIncidenceArray`

AbstractIncidenceArray

Description

An abstract class for storing an array. It has an array of data arr, which it is responsible for storing. For arrays with particular metadata, consider extending this class. However, if the class is not truly an array, consider extending FrameData.

Fields

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The size of the array.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

Methods

addSlices(number,dimension=2,mutate=TRUE): This method **must** be extended. Extend a dimension by adding extra indices to the end.

number - How many slices to add.
dimension - Which dimension should slices be added to.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

apply(FUNC,dimension=c(1,2)): This method **must** be extended. Apply a function to each slice.

FUNC - The function to apply
dimension - The dimension(s) to hold fixed.

Arguments

Value An IncidenceArray with dimension equal to `self$dims[dimension]`

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

subset(...,mutate=TRUE): This method **must** be extended. Take a subset of the matrix.

...	-	dimensional indices in order.
mutate	-	Whether to modify the existing object, or return a modified copy.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string	-	The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : `ArrayData`

Examples

```
require(abind)
SampleIncidenceArray  <- R6Class(
  inherit= AbstractIncidenceArray,
  public = list(
    initialize = function(data,dims = dim(data),dnames,dimData,metaData){
      private$.arr <- array(data,dims)
      if(!missing(dnames)){
        dimnames(private$.arr) <- dnames
        private$.dnames <- dnames
      }
      private$.dims = dim(data)
      private$.ndim = length(dim(data))
      if(!missing(dimData)){
        self$dimData <- dimData
      } else {
        self$dimData <- rep(list(list()),self$ndim)
      }
      if(!missing(metaData)){
        self$metaData <- metaData
      }
    },
    addSlices = function(number,dimension,mutate=TRUE){
      if(!mutate){
        return(self$clone(TRUE)$addSlices(number,dimension))
      }
```

```
      dims <- private$.dims
      dims[dimension] <- number
      private$.arr <-
        abind(
          private$.arr,
          array(NA,dims),
          along=dimension
        )
      private$.dims = dim(private$.arr)
      private$.dnames = dimnames(private$.arr)
    },
    subset = function(...,mutate=TRUE){
      if(!mutate){
        return(self$clone(TRUE)$subset(number,dimension))
      }
      private$.arr = private$.arr[...]
      private$.dnames = dimnames(private$.arr)
      private$.dims = dim(private$.arr)
      self$dimData <-
        mapply(
          index = list(...),
          obj = self$dimData,
          function(index,obj){
            lapply(obj,function(x){x[index]})
          },
          SIMPLIFY = FALSE
        )
    }
  )
)

data = SampleIncidenceArray$new(array(1:27,c(3,3,3)))
data$addSlices(1,1)
data$addSlices(2,2)
data$addSlices(3,3)
data$arr
data$dims
data$subset(1:2,1:2,1:2)
```

`AbstractIncidenceMatrix`

AbstractIncidenceMatrix

Description

An abstract class for storing an actual matrix. It has an actual matrix of data mat, which it is responsible for storing. For creating matrices with particular metadata, consider extending IncidenceMatrix instead of this class. Extend this class if you have data which can be thought of as a matrix, but that is not its true form. ## TODO: Include an example of this.

Fields

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the columns of the data.

Methods

addColumns(columns,mutate=TRUE): This method **must** be extended. This function adds empty columns to the right side of the data.

columns - The number of columns to add.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method wil

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

addRows(rows,mutate=TRUE): This method **must** be extended. This function adds empty rows to the data.

rows - The number of rows to add.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

diff(lag=1,mutate=TRUE): This method **must** be extended. This function replaces the matrix value at column i with the difference. between the values at columns i and (i-lag).

lag - How far back to diff. Defaults to 1.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

head(k,direction,mutate=TRUE...): This method **must** be extended. Select the first k slices of the data in dimension direction.

k - The number of slices to keep.
direction - The dimension to take a subset of. 1 for row, 2 for column.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method w

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

lag(indices,mutate=TRUE,na.rm=FALSE): This method **must** be extended. This function replaces the current matrix with a new matrix with one column for every column, and a row for every row/index combination. The column corresponding to the row and index will have the value of the original matrix in the same row, but index columns previous. This shift will introduce NAs where it passes off the end of the matrix.

indices - A sequence of lags to use as part of the data. Note that unless this list contains 0, the data will all be shifted back
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will
na.rm - Whether to remove NA values generated by walking off the edge of the matrix.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

mutate(rows,cols,data): This method **must** be extended. This function is a way to modify the data as though it were a matrix.

rows - Which rows to modify. These can be numeric or names.
cols - Which cols to modify. These can be numeric or names.
data - The data to change the chosen values to. It needs to be the right shape.

Arguments

scale(f,mutate=TRUE): This method **must** be extended. This function rescales each element of our object according to f

f - a function which takes in a number and outputs a rescaled version of that number
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

subset(rows,cols,mutate=TRUE...): This method **must** be extended. Select the data corresponding to the rows `rows` and the columns `columns`. `rows` and `columns` can be either numeric or named indices.

rows - An row index or list of row indices which can be either numeric or named.
cols - An column index or list of column indices which can be either numeric or named.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

tail(k,direction,mutate=TRUE...): This method **must** be extended. Select the last `k` slices of the data in dimension `direction`.

k - The number of slices to keep.
direction - The dimension to take a subset of. 1 for row, 2 for column.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method w

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : MatrixData

Is inherited by : IncidenceMatrix

Examples

```
IncidenceMatrix <- R6Class(
  classname = "IncidenceMatrix",
  inherit = AbstractIncidenceMatrix,
  public = list(
    initialize = function(
      data=matrix(),
      metaData=list(),
      rowData=list(),
      colData=list()
    ){
      if(Reduce(
        '&&',
        c('MatrixData','DataContainer','Generic','R6') %in% class(data))
      ){
        private$.mat <- data$mat
        private$.metaData <- data$metaData
        private$.nrow <- data$nrow
        private$.ncol <- data$ncol
        private$.rnames <- data$rnames
        private$.cnames <- data$cnames
        private$.rowData <- data$rowData
        private$.colData <- data$colData
        private$.metaData <- data$metaData
        private$.cellData <- data$cellData
      }
      else{
        rtoggle = FALSE
        ctoggle = FALSE
        try({
          rnames <- dimnames(data)[[1]]
          rtoggle = TRUE
        })
        try({
          cnames <- dimnames(data)[[2]]
          ctoggle = TRUE
        })
        if(!private$checkType(name='.mat',val=data,type='private')){
          data <- as.matrix(data)
          if(rtoggle){
            rownames(data) = rnames
          }
          if(ctoggle){
            colnames(data) = cnames
```

```
      }
    }
    if(!private$checkType(name='.mat',val=data,type='private')){
      stop(paste(
        "invalid data of type",
        paste(class(data),collapse=','),
        "expected",
        paste(class(private$.mat),collapse = ',')
      ))
    }
    if(length(dim(data)) > 2){
      stop("The matrix is not intended to hold things with more than 3 dimensions.")
    }
    ndim = dim(data)
    private$.nrow = ndim[[1]]
    private$.ncol = ndim[[2]]
    private$.rnames = rownames(data)
    private$.cnames = colnames(data)
    private$.mat <- 0+data
    self$rowData <- rowData
    self$colData <- colData
    self$metaData <- metaData
  }
},
subset = function(rows,cols,mutate=TRUE){
  if('subset' %in% private$.debug){
    browser()
  }
  if(!mutate){
    temp = self$clone(TRUE)
    temp$subset(rows,cols,mutate=TRUE)
    return(temp)
  }
  if(missing(rows) && missing(cols)){
  }
  else if(missing(rows)){
    private$.mat = self$mat[,cols,drop=FALSE]
    if(length(private$.colData) > 0){
      private$.colData <- lapply(
        private$.colData,function(x){x[cols,drop=FALSE]}
      )
    }
  }
  else if(missing(cols)){
    private$.mat = self$mat[rows,,drop=FALSE]
    if(length(private$.rowData) > 0){
      private$.rowData <- lapply(
        private$.rowData,function(x){x[rows,drop=FALSE]}
      )
    }
  }
  else{
    private$.mat = self$mat[rows,cols,drop=FALSE]
```

```
    if(length(private$.rowData)>0){
      private$.rowData <- lapply(
        private$.rowData,function(x){x[rows,drop=FALSE]}
      )
    }
    if(length(private$.colData)>0){
      private$.colData <- lapply(
        private$.colData,function(x){x[cols,drop=FALSE]}
      )
    }
  }
  private$.nrow = nrow(private$.mat)
  private$.rnames = rownames(private$.mat)
  private$.ncol = ncol(private$.mat)
  private$.cnames = colnames(private$.mat)
},
head = function(k,direction=2){
  if('head' %in% private$.debug){
    browser()
  }
  if(k>dim(private$.mat)[[direction]]){
    stop("The size of the head is too large.")
  }
  indices = 1:k
  if(direction==1){
    private$.mat = self$mat[indices,,drop=FALSE]
    if(length(private$.rowData)>0){
      for(i in 1:length(private$.rowData)){
        private$.rowData[[i]] = private$.rowData[[i]][indices,drop=FALSE]
      }
    }
  }
  else if(direction==2){
    private$.mat = self$mat[,indices,drop=FALSE]
    if(length(private$.colData)>0){
      for(i in 1:length(private$.colData)){
        private$.colData[[i]] = private$.colData[[i]][indices,drop=FALSE]
      }
    }
  }
  else{
    stop("This direction is not allowed.")
  }
  private$.nrow = nrow(private$.mat)
  private$.ncol = ncol(private$.mat)
  private$.cnames = colnames(private$.mat)
  private$.rnames = rownames(private$.mat)
},
tail = function(k,direction=2){
  if('tail' %in% private$.debug){
    browser()
  }
  if(k>dim(private$.mat)[[direction]]){
```

```
    stop("The size of the tail is too large.")
  }
  indices = (dim(self$mat)[[direction]]-k+1):dim(self$mat)[[direction]]
  if(direction==1){
    private$.mat = self$mat[indices,,drop=FALSE]
    if(length(private$.rowData)>0){
      for(i in 1:length(private$.rowData)){
        private$.rowData[[i]] = private$.rowData[[i]][indices,drop=FALSE]
      }
    }
  }
  else if(direction==2){
    private$.mat = self$mat[,indices,drop=FALSE]
    if(length(private$.colData)>0){
      for(i in 1:length(private$.colData)){
        private$.colData[[i]] = private$.colData[[i]][indices,drop=FALSE]
      }
    }
  }
  else{
    stop("This direction is not allowed.")
  }
  private$.nrow = nrow(private$.mat)
  private$.ncol = ncol(private$.mat)
  private$.cnames = colnames(private$.mat)
  private$.rnames = rownames(private$.mat)
},
lag = function(indices,mutate=TRUE,na.rm=FALSE){
  if('lag' %in% private$.debug){
    browser()
  }
  if(mutate==FALSE){
    tmp = self$clone(TRUE)
    tmp$lag(indices=indices,mutate=TRUE,na.rm=na.rm)
    return(tmp)
  }
  if((1+max(indices)) > self$ncol){
    stop("We cannot go further back than the start of the matrix")
  }
  numLags = length(indices)
  oldNrow = self$nrow
  if(is.null(rownames(private$.mat))){
    rownames(private$.mat) = 1:(dim(private$.mat)[[1]])
  }
  rownames = replicate(numLags,rownames(private$.mat))
  colnames = colnames(private$.mat)
  private$.mat <- 0+array(self$mat,c(dim(self$mat),numLags))
  if(numLags <= 0){
    stop("indices must be nonempty for the calculation of lags to make sense.")
  }
  for(lag in 1:numLags){
    private$.mat[,(1+indices[[lag]]):self$ncol,lag] <-
      private$.mat[,1:(self$ncol-indices[[lag]]),lag]
```

```
    if(indices[[lag]] > 0){
      private$.mat[,1:(indices[[lag]]),lag] = NA
    }
  }
  private$.mat = aperm(private$.mat,c(1,3,2))
  private$.mat = matrix(private$.mat,self$nrow*numLags,self$ncol)
  lagnames = t(replicate(self$nrow,paste('L',indices,sep='')))
  rownames(private$.mat) <-
    as.character(paste(lagnames,"R",rownames,sep=''),numLags*self$nrow)
  colnames(private$.mat) <- colnames
  private$.nrow = self$nrow * numLags
  private$.rnames = rownames(private$.mat)
  if(length(private$.rowData) > 0){
    private$.rowData <- lapply(
      private$.rowData,
      function(x){
        c(unlist(recursive=FALSE,lapply(1:numLags,function(y){x})))
      }
    )
  }
  if(na.rm==T){
    self$subset(cols=!apply(private$.mat,2,function(x){any(is.na(x))}))
  }
},
scale = function(f,mutate=TRUE){
  if('scale' %in% private$.debug){
    browser()
  }
  if(!mutate){
    tmp = self$clone(TRUE)
    tmp$scale(f=f,mutate=TRUE)
    return(tmp)
  }
  private$.mat[] = f(private$.mat[])
},
diff = function(lag = 1,mutate=TRUE){
  if('diff' %in% private$.debug){
    browser()
  }
  if(lag == 0){
    if(!is.null(private$.rnames)){
      rownames(private$.mat) =
        paste("D",lag,"R",private$.rnames,sep='')
      private$.rnames = rownames(private$.mat)
    } else {
      rownames(private$.mat) =
        paste("D",lag,"R",1:private$.nrow,sep='')
      private$.rnames = rownames(private$.mat)
    }
    return()
  }
  if(lag < 0){
    stop("Lag should be non-negative.")
```

```
  }
  if(!mutate){
    tmp = self$clone(TRUE)
    tmp$diff(lag=lag,mutate=TRUE)
    return(tmp)
  }
  private$.mat <-
    self$mat - self$lag(indices=lag,mutate=FALSE,na.rm=FALSE)$mat
  if(!is.null(private$.rnames)){
    rownames(private$.mat) =
      paste("D",lag,"R",private$.rnames,sep='')
    private$.rnames = rownames(private$.mat)
  } else {
    rownames(private$.mat) =
      paste("D",lag,"R",1:private$.nrow,sep='')
    private$.rnames = rownames(private$.mat)
  }
},
addColumns = function(columns){
  if('addColumns' %in% private$.debug){
    browser()
  }
  if(columns == 0){
    return()
  }
  cbind(private$.mat , matrix(NA,private$.nrow,columns)) -> private$.mat
  private$.ncol = ncol(private$.mat)
  if(!is.null(private$.cnames)){
    colnames(private$.mat) = c(private$.cnames,replicate(columns,"NA"))
    private$.cnames = colnames(private$.mat)
  }
  if(length(private$.colData) > 0){
    private$.colData <- lapply(
      private$.colData,
      function(x){
        c(x,replicate(columns,NA))
      }
    )
  }
},
addRows = function(rows){
  if('addRows' %in% private$.debug){
    browser()
  }
  if(rows == 0){
    return()
  }
  rbind(private$.mat , matrix(NA,rows,private$.ncol)) -> private$.mat
  private$.nrow = nrow(private$.mat)
  if(!is.null(private$.rnames)){
    rownames(private$.mat) = c(private$.rnames,replicate(rows,"NA"))
    private$.rnames = rownames(private$.mat)
  }
```

```
    if(length(private$.rowData) > 0){
      private$.rowData <- lapply(
        private$.rowData,
        function(x){
          c(x,replicate(rows,NA))
        }
      )
    }
  },
  mutate = function(rows,cols,data){
    if('mutate' %in% private$.debug){
      browser()
    }
    data = as.matrix(data)
    if(missing(rows)){
      rows = 1:private$.nrow
      if(!(is.null(private$.cnames) || is.null(colnames(data)))){
        private$.cnames[cols] = colnames(data)
        colnames(private$.mat) = private$.cnames
      }
    }
    if(missing(cols)){
      cols = 1:private$.ncol
      if(!(is.null(private$.rnames) || is.null(rownames(data)))){
        private$.rnames[rows] = rownames(data)
        rownames(private$.mat) = private$.rnames
      }
    }
    if(is.null(dim(data))){
      stop("Not yet implemented for non-matrixlike objects")
    }
    if(length(dim(data)) > 2){
      stop("There are too many dimensions in data.")
    }
    if(length(dim(data)) == 2){
      private$.mat[rows,cols] = data
    }
  }
),
active = list(
  mat = function(value){
    "The matrix of data."
    if('mat' %in% private$.debug){
      browser()
    }
    if(missing(value)){
      return(private$.mat)
    }
    stop(
      "Do not write directly to the mat. Either use methods to modify the mat,
       or create a new instance."
    )
  },
```

```
colData = function(value){
  "The metaData associated with column in the matrix"
  if('colData' %in% private$.debug){
    browser()
  }
  if(missing(value)){
    if(length(private$.colData) > 0){
      for(i in 1:length(private$.colData)){
        if(private$.ncol != length(private$.colData[[i]])){
          stop("If you alter the matrix, please also edit the column metaData.")
        }
      }
    }
    return(private$.colData)
  }
  if(class(value) != 'list'){
    stop("Column metaData should be a list of lists.")
  }
  if(length(value)>0){
    for(i in 1:length(value)){
      if(
        Reduce(
          '&&',
          class(value[[i]]) !=
            c(
              'list',
              'character',
              'numeric',
              'integer',
              'logical',
              'raw',
              'complex'
            )
        )
      ){
        if(dim(as.matrix(value[[i]]))[[1]] != private$.ncol){
          stop(paste(
            'The ',
            i,
            'th element of column metaData does not have one element for',
            'each column.',
            sep=''
          ))
        }
      }
      else{
        if(length(value[[i]])!=private$.ncol){
          stop(paste(
            'The ',
            i,
            'th element of column metaData does not have one element for',
            'each column.',
            sep=''
```

```
          ))
        }
      }
    }
  }
  private$.colData <- value
  if(length(private$.colData) > 0){
    for(i in 1:length(private$.colData)){
      names(private$.colData[[i]]) <- colnames(self$mat)
    }
  }
},
rowData = function(value){
  "The metaData associated with rows in the matrix"
  if('rowData' %in% private$.debug){
    browser()
  }
  if(missing(value)){
    if(length(private$.rowData) > 0){
      for(i in 1:length(private$.rowData)){
        if(private$.nrow != length(private$.rowData[[i]])){
          stop("If you alter the matrix, please also edit the row metaData.")
        }
      }
    }
    return(private$.rowData)
  }
  if(class(value) != 'list'){
    stop("row metaData should be a list of lists.")
  }
  if(length(value) > 0){
    for(i in 1:length(value)){
      if(
        Reduce('&&',
          class(value[[i]]) !=
            c(
              'list',
              'character',
              'numeric',
              'integer',
              'logical',
              'raw',
              'complex'
            )
        )
      ){
        if(dim(as.matrix(value[[i]]))[[1]] != private$.nrow){
          stop(paste(
            'The ',
            i,
            'th element of row metaData does not have one element for each',
            'row.',
            sep=''
```

```
              ))
            }
          }
          else{
            if(length(value[[i]])!=private$.nrow){
              stop(paste(
                'The ',
                i,
                'th element of row metaData does not have one element for each',
                'row.',
                sep=''
              ))
            }
          }
        }
      }
      private$.rowData <- value
      if(length(private$.rowData)>0){
        for(i in 1:length(value)){
          names(private$.rowData[[i]]) <- rownames(self$mat)
        }
      }
    }
  )
)
```

`AbstractObservationList`

AbstractObservationList

Description

A class for storing data in multiple interrelated tables. Each table relates to the other tables through keys.

Fields

aggregate A function used to combine covariates of the same key/val pair.

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The size of the array.

frame Long form data.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

formArray(val,...,metaData=list(),dimData=list()): This method **must** be extended. In order to use an ObservationList as an ArrayData, you need to select which columns to use to form the dimensions of the array. Optionally, you can also assign some of the columns to be associated with each dimension (or cell). Note that `aggregate` is used to determine how to deal with multiple observations associated with a particular grouping.

val - The attribute of `frame` to use for the values of the array (must `aggregate_` to a numeric type)
... - Column names of columns which, in order should form the dimensions of the array
metaData - The attribute(s) of `frame` to store in metaData so they can be accessed by methods expecting a MatrixData ob

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : FrameData

Examples

```
require(dplyr)
ObservationList<- R6Class(
  classname= "ObservationList",
  inherit = AbstractObservationList,
  private = list(
    .aggregate = NULL,
    .frame = data_frame(),
    aCurrent = FALSE,
    .aDims = list(),
    .aVal = '',
    .slice = 1,
    .aDimData = list(),
    na.rm = FALSE,
    updateArray = function(na.rm = private$na.rm){
      if('updateArray' %in% private$.debug){
        browser()
      }

      if(length(self$aDims) == 0){
        warning("Form array with formArray first")
        private$.arr = array(as.numeric(NA),c(0,0))
        private$.ndim = 2
        private$.dims = c(0,0)
        private$.dimData = list(list(),list())
        private$.dnames = list()
        return()
      }
      if(nrow(private$.frame) == 0){
        warning("The frame is empty.")
        private$.arr = array(as.numeric(NA),c(1:self$aDims*0))
        private$.ndim = length(self$aDims)
        private$.dims = 1:length(self$aDims)*0
        private$.dimData = lapply(self$aDims,list())
        private$.dnames = list()
        return()
      }
      private$.dnames = lapply(self$aDims,function(name){
        as.character(unique(private$.frame[[name]]))
      })
      private$.dims = sapply(private$.dnames,length)
      private$.ndim = length(private$.dims)
      private$.arr <- private$.frame %>%
        select_(
          .dots = setNames(
            c(unlist(self$aDims),self$aVal),
```

```
          nm=c(unlist(self$aDims),self$aVal)
        )
      ) %>%
      group_by_(.dots=setNames(self$aDims,NULL)) %>%
      self$aggregate() %>%
      ungroup() %>%
      acast(as.formula(paste(self$aDims,collapse='~')),value.var=self$aVal)
    mode(private$.arr) = 'numeric'
    private$.dnames = dimnames(private$.arr)
    if(class(self$aDimData) == 'list'){
      if(length(self$aDimData) < length(self$aDims)){
        self$aDimData[length(self$aDims)] = list(NULL)
      }
    } else {
      stop("array metadata should be a list of columns in the data frame.")
    }
    private$.dimData = mapply(dim=self$aDims,data=self$aDimData,function(dim,data){
      self$frame %>%
        group_by_(dim) %>%
        self$aggregate() %>%
        ungroup() %>%
        select_(.dots=data) %>%
        as.list()
    })
    if(length(private$.dimData) < private$.ndim){
      private$.dimData = lapply(1:private$.ndim,function(x){
        if(length(private$.dimData >= x)){
          private$.dimData[[x]]
        } else {
          list()
        }
      })
    }
    private$aCurrent <- TRUE
  }
),
public = list(
  initialize = function(data=data_frame(),...){
    self$frame <- as_data_frame(data)
    self$formArray(...)
    private$.aggregate = function(input_data){
      if('aggregate' %in% private$.debug){
        browser()
      }
      grouping = groups(input_data)
      input_data %>%
        input_data <- summarize_all(funs(
          sum = if(is.numeric(.) || is.logical(.)){
            sum(.,na.rm=private$na.rm)
          } else{NA},
          unique = if(length(unique(.))==1){unique(.)} else{list(unique(.))}))
      input_data <- input_data %>%
        group_by_(.dots=grouping)
```

```
    if('sum' %in% names(input_data)){
      input_data <- input_data %>%
        rename_(.dots=setNames('sum',paste(private$.aVal,'_sum',sep='')))
    }
    if('unique' %in% names(input_data)){
      input_data <- input_data %>%
        rename_(
          .dots=setNames('unique',paste(private$.aVal,'_unique',sep=''))
        )
    }
    column_names = c(
      unlist(
        sapply(private$.aDimData,function(x){paste(x,"_unique",sep='')})
      ),
      paste(private$.aVal,'_sum',sep=''))
    column_names = column_names[column_names %in% names(input_data)]
    input_data %>%
      select_(.dots=column_names) ->
      input_data
    input_data %>%
      rename_(.dots=setNames(
        names(input_data)[grepl(names(input_data),pattern='_unique')],
        lapply(
          names(input_data)[grepl(names(input_data),pattern='_unique')],
          function(x){substr(x,1,nchar(x)-7)}
        )
      )) %>%
      rename_(.dots=setNames(
        names(input_data)[grepl(names(input_data),pattern='_sum')],
        lapply(
          names(input_data)[grepl(names(input_data),pattern='_sum')],
          function(x){substr(x,1,nchar(x)-4)}
        )
      )) %>%
      return()
  }
},
formArray = function(...,val,dimData=list(),metaData=list()){
  if('formArray' %in% private$.debug){
    browser()
  }
  if(missing(val)){
    if(length(list(...))==0){
      return()
    } else{
      stop("val must be supplied in order to form the incidenceArray")
    }
  }
  private$aCurrent=FALSE
  private$.dnames = NULL
  self$aDims = list(...)
  self$aVal = val
  self$aDimData = dimData
```

```
    metaDataKeys <-
      names(private$.metaData)[!(names(private$.metaData) %in% names(metaData))]
    self$metaData = c(private$.metaData[metaDataKeys],metaData)
  }
),
active = list(
  frame = function(value){
    if('frame' %in% private$.debug){
      browser()
    }
    private$aCurrent = FALSE
    if(!missing(value)){
      private$.rnames = NULL
      private$.cnames = NULL
    }
    private$defaultActive('.frame','private',value)
  },
  arr = function(value){
    if('arr' %in% private$.debug){
     browser()
    }
    if(missing(value)){
      if(private$aCurrent == FALSE){
        private$updateArray()
      }
      return(private$.arr)
    }
    stop("Do not write directly to the array.")
  },
  dims = function(value){
    if('dims' %in% private$.debug){
     browser()
    }
    if(missing(value)){
      if(private$aCurrent == FALSE){
        private$updateArray()
      }
      return(private$.dims)
    }
    stop("Do not write directly to the dimensions.")
  },
  ndim = function(value){
    if('ndim' %in% private$.debug){
     browser()
    }
    if(missing(value)){
      if(private$aCurrent == FALSE){
        private$updateArray()
      }
      return(private$.ndim)
    }
    stop("Do not write directly to the number of dimensions.")
  },
```

```
dimData = function(value){
  if('dimData' %in% private$.debug){
   browser()
  }
  if(private$aCurrent == FALSE){
    private$updateArray()
  }
  if(missing(value)){
    return(private$.dimData)
  }
  if(is.null(value)){
    private$.dimData=NULL
  } else if(class(value) == 'list'){
    nval = length(value)
    if(nval == 0){
      private$.dimData = value
    } else if(nval <= private$.ndim){
      if(all(mapply(function(self,other){
        is.null(other) ||
        all(sapply(other,function(x){self==length(x)}))
      },
      self = private$.dims[1:nval],
      other=value
      ))){
        private$.dimData[!sapply(value,is.null)] = value[!sapply(value,is.null)]
      } else {
        stop("The dimensions don't match up.")
      }
    } else {
      stop("Invalid number of dimensions.")
    }
  } else {
    stop(paste(
      "Not sure how to make dimension metaData from object of class",
      class(value)
    ))
  }
},
dnames = function(value){
  if('dnames' %in% private$.debug){
   browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    return(private$.dnames)
  }
  stop("Do not write directly to the dimension names.")
},
nrow = function(value){
  if('nrow' %in% private$.debug){
   browser()
```

```
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    return(private$.dims[1])
  }
  stop("Do not write directly to the number of rows.")
},
ncol = function(value){
  if('ncol' %in% private$.debug){
   browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    return(private$.dims[2])
  }
  stop("Do not write directly to the number of columns.")
},
rnames = function(value){
  if('rnames' %in% private$.debug){
   browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    return(private$.dnames[[1]])
  }
  stop("Do not write directly to the row names.")
},
cnames = function(value){
  if('cnames' %in% private$.debug){
   browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    return(private$.dnames[[2]])
  }
  stop("Do not write directly to the column names.")
},
colData = function(value){
  if('colData' %in% private$.debug){
   browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
```

```
    return(self$dimData[[2]])
  }
  self$dimData[[2]] <- value
},
rowData = function(value){
  if('rowData' %in% private$.debug){
   browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    return(self$dimData[[1]])
  }
  self$dimData[[1]] <- value
},
aDims = function(value){
  if('aDims' %in% private$.debug){
    browser()
  }
  if(missing(value)){
    return(private$.aDims)
  }
  lapply(value,function(value){
    if(!all(value %in% colnames(private$.frame))){
      stop(paste(value,"is not a column of the frame"))
    }
  })
  private$.aDims = value
  private$aCurrent = FALSE
},
aVal = function(value){
  if('aVal' %in% private$.debug){
    browser()
  }
  if(missing(value)){
    return(private$.aVal)
  }
  if(!(value %in% colnames(private$.frame))){
    stop(paste(value,"is not a column of the frame"))
  }
  private$.aVal = value
  private$aCurrent = FALSE
},
aDimData = function(value){
  if('aDimData' %in% private$.debug){
    browser()
  }
  if(missing(value)){
    return(private$.aDimData)
  }
  lapply(value,function(value){
    for(val in value){
```

```
      if(!(val %in% colnames(private$.frame))){
        stop(paste(val,"is not a column of the frame"))
      }
    }
  })
  private$.aDimData = value
  private$aCurrent = FALSE
},
mat = function(value){
  if('mat' %in% private$.debug){
    browser()
  }
  if(missing(value)){
    if(private$aCurrent == FALSE){
      private$updateArray()
    }
    if(private$.ndim == 2){
      return(as.matrix(private$.arr))
    }
    return(apply(private$.arr,c(1,2),function(x){x[slice]}))
  }
  stop(paste(
    "Do not write directly to the mat, because it is automatically",
    "calculated.  The Observation List is called frame"
  ))
},
slice = function(value){
  if(missing(value)){
    return(private$.slice)
  }
  if(any(c(1,1,value) > self$dims)){
    stop("Value must be between 1 and length in that dimension.")
  }
  private$.slice = matrix(value,1)
},
aggregate = function(value){
  if(missing(value)){
    return(private$.aggregate)
  }
  private$aCurrent = FALSE
  if(class(value) != 'function'){
    stop(paste(
      "Not a function.  aggregate should be a function taking a single",
      "data_frame argument called input_data"
    ))
  }
  if(length(names(formals(fun=value))) != 1){
    stop(paste(
      "Not a valid function for aggregation.  A valid aggregation function",
      "must take a single data_frame argument."
    ))
  }
  private$.aggregate = value
```

```
    }
  )
)
```

`AbstractRelationalTables`
AbstractRelationalTables

Description

A class for storing data in multiple interrelated tables. Each table relates to the other tables through keys.

Fields

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The size of the array.

frame The data frame this class is responsible for.

keys A list containing the primary keys for each table

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

tables A list of tables of data

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : RelationalData

Examples

```
library(dplyr)
library(reshape2)
SampleRelationalTables <- R6Class(
  inherit = AbstractRelationalTables,
  public = list(
    initialize = function(...){
      private$.tables = list(...)
      if(!all(sapply(private$.tables,function(x){is.data.frame(x)}))){
        stop("All arguments must be data frames")
      }
    },
    updateFrame = function(){
      private$.frame = Reduce(x = private$.tables,f = left_join)
    },
    updateArray = function(){
      val <- names(self$frame)[1]
      dims <- names(self$frame[2:ncol(self$frame)])
      private$.arr <- self$frame %>%
        group_by_(.dots=setNames(dims,NULL)) %>%
        summarize_all(sum) %>%
        ungroup() %>%
        acast(as.formula(paste(dims,collapse='~')),value.var=val)
      mode(private$.arr) = 'numeric'
      private$.dims = dim(private$.arr)
      private$.nrow = private$.dims[1]
```

```
      private$.ncol = private$.dims[2]
      private$.ndim = length(private$.dims)
      private$.dnames = dimnames(private$.arr)
    }
  ),
  active = list(
    mat = function(value){
      if(self$ndim <= 2){
        return(self$arr)
      }
    return(extract(private$.arr,indices = rep(self$ndim-2,x=1),dims = 3:self$ndim,drop=TRUE))
    }
  )
)
```

`AbstractSimulatedIncidenceMatrix`

AbstractSimulatedIncidenceMatrix

Description

This class stores a number of simulations each of which contains the same data as an IncidenceMatrix.

Fields

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The names of dimensions of the data.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

nsim The number of simulations in `self$simulaions`

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

sample An IncidenceMatrix sampled from the simulations.

simulations The array of simulations. This is another name for 'arr'.

Methods

addColumns(columns,mutate=TRUE): This method **must** be extended. This function adds empty columns to the right side of the data.

columns - The number of columns to add.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method wi

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

addError(type): This method **must** be extended. Add error of a particular type to the data.

type - What sort of error to add.

Arguments

addRows(rows,mutate=TRUE): This method **must** be extended. This function adds empty rows to the right side of the data.

rows - The number of rows to add.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

diff(lag=1,mutate=TRUE): This method **must** be extended. This function replaces the matrix value at column i with the difference. between the values at columns i and (i-lag).

lag - How far back to diff. Defaults to 1.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

head(k,direction,mutate=TRUE...): This method **must** be extended. Select the first k slices of the data in dimension direction.

k - The number of slices to keep.
direction - The dimension to take a subset of. 1 for row, 2 for column.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method w

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

lag(indices,mutate=TRUE): This method **must** be extended. This function replaces the current matrix with a new matrix with one column for every column, and a row for every row/index combination. The column corresponding to the row and index will have the value of the original matrix in the same row, but index columns previous. This shift will introduce NAs where it passes off the end of the matrix.

indices - A sequence of lags to use as part of the data. Note that unless this list contains `0`, the data will all be shifted back
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

mutate(rows,cols,data): This method **must** be extended. This function is a way to modify the data as though it were a matrix. `self$mutate(row,col,data)` is equivalent to `self$mat[row,col] <-data`.

rows - Which rows to modify. These can be numeric or names.
cols - Which cols to modify. These can be numeric or names.
data - The data to change the chosen values to. It needs to be the right shape.

Arguments

scale(f,mutate=TRUE): This method **must** be extended. This function rescales each element of our object according to f

f - a function which takes in a number and outputs a rescaled version of that number
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

subsample(simulations,mutate=TRUE...): This method **must** be extended. Select only some of the simulations.

simulations - An index or list of column indices which simulations to keep.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

subset(rows,cols,mutate=TRUE...): This method **must** be extended. Select the data corresponding to the rows `rows` and the columns `columns`. `rows` and `columns` can be either numeric or named indices.

rows - An row index or list of row indices which can be either numeric or named.
cols - An column index or list of column indices which can be either numeric or named.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

summarize(FUNC): This method **must** be extended. Apply a function FUNC to every simulation elementwise.

FUNC - a function which should be applied to every simulation. It should reduce each simulation to a single number.

Arguments

Value A MatrixData where `return$mat` is `FUNC` applied to every simulation.

tail(k,direction,mutate=TRUE...): This method **must** be extended. Select the last `k` slices of the data in dimension `direction`.

k - The number of slices to keep.

direction - The dimension to take a subset of. 1 for row, 2 for column.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method w

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : `ArrayData`

Examples

```
SimulatedIncidenceMatrix <- R6Class(
  classname = "SimulatedIncidenceMatrix",
  inherit = AbstractSimulatedIncidenceMatrix,
  private = list(
    .ndim = 3,
    ncore = 1,
    .sample = 1,
    parallelEnvironment = NULL
  ),
  public = list(
    initialize = function(data=MatrixData$new(),nsim=1){
      if('AbstractSimulatedIncidenceMatrix' %in% class(data)){
        private$.arr= data$simulations
        private$.metaData = data$metaData
        private$.dimData = data$dimData
        private$.dnames = data$dnames
        private$.dims = data$dims
        private$.ndim = length(self$dims)
        return()
      }
      else if('list' %in% class(data)){
        if('MatrixData' %in% class(data[[1]])){
          if((!missing(nsim)) && (nsim != length(data))){
            stop("nsim is not used for list data.")
          }
          private$.dims = c(data[[1]]$nrow,data[[1]]$ncol,length(data))
          private$.arr = array(NA,c(self$nrow,self$ncol,self$nsim))
          if(self$nsim > 0){
            for(i in 1: self$nsim){
```

```
        private$.arr[,,i] = data[[i]]$mat
      }
    }
    data = data[[1]]
    private$.metaData = data$metaData
    private$.dimData = list(data$rowData,data$colData)
    if(!is.null(data$rnames)){
      if(!is.null(data$cnames)){
        private$.dnames = list(data$rnames,data$cnames,NULL)
      } else{
        private$.dnames = list(data$rnames,NULL,NULL)
      }
    } else if(!is.null(data$cnames)){
      private$.dnames = list(NULL,data$cnames,NULL)
    } else {
      private$.dnames = NULL
    }
    dimnames(private$.arr) = private$.dnames
    private$.dims = c(data$nrow,data$ncol,nsim)
    private$.ndim = 3
    return()
  }
  else{
    stop("Not yet implemented")
  }
} else if('MatrixData' %in% class(data)){
  private$.arr= array(data$mat,c(data$nrow,data$ncol,nsim))
  private$.metaData = data$metaData
  private$.dimData = list(data$rowData,data$colData)
  if(!is.null(data$rnames)){
    if(!is.null(data$cnames)){
      private$.dnames = list(data$rnames,data$cnames,NULL)
    } else{
      private$.dnames = list(data$rnames,NULL,NULL)
    }
  } else if(!is.null(data$cnames)){
    private$.dnames = list(NULL,data$cnames,NULL)
  } else {
    private$.dnames = NULL
  }
  dimnames(private$.arr) = private$.dnames
  private$.dims = c(data$nrow,data$ncol,nsim)
  private$.ndim = 3
  return()
}
else{
  stop("Input data is not a valid type to make a SimulatedIncidenceMatrix")
}
stop("This is currently broken.")
rownames(private$.arr) <- rownames(data[[1]]$mat)
colnames(private$.arr) <- colnames(data[[1]]$mat)
private$.dimData = list(data$rowData,data$colData,NULL)
private$.metaData = data$metaData
```

```
  if(!is.null(data$rnames)){
    if(!is.null(data$cnames)){
      private$.dnames = list(data$rnames,data$cnames,NULL)
    } else{
      private$.dnames = list(data$rnames,NULL,NULL)
    }
  } else if(!is.null(data$cnames)){
    private$.dnames = list(NULL,data$cnames,NULL)
  } else {
    private$.dnames = NULL
  }
  private$.dims = c(data$nrow,data$ncol,nsim)
},
mean = function(){
  "Compute the mean across simulations"
  if('mean' %in% private$.debug){
    browser()
  }
  return(IncidenceMatrix$new(apply(self$simulations,c(1,2),mean)))
},
median = function(){
  if('median' %in% private$.debug){
    browser()
  }
  return(IncidenceMatrix$new(apply(self$arr,c(1,2),median)))
},
addError = function(type,rows,cols,mutate = TRUE){
  if('addError' %in% private$.debug){
    browser()
  }
  if(missing(rows)){
    rows = 1:self$nrow
  }
  if(missing(cols)){
    cols = 1:self$ncol
  }
  if(type=='Poisson'){
    private$.arr[rows,cols,] =
      rpois(length(rows)*length(cols)*self$nsim,self$arr[rows,cols,])
  } else{
    stop("Not yet implemented")
  }
},
subsample = function(simulations,mutate=TRUE){
  if('subsample' %in% private$.debug){
    browser()
  }
  if(!mutate){
    rc = self$clone(TRUE)
    rc$subsample(simulations,mutate=TRUE)
    return(rc)
  }
  if(
```

```
  (min(simulations) < 0) ||
    (max(simulations) > self$nsim) ||
    any(round(simulations) != simulations)
  ){
    stop("simulations out of bounds.")
  }
  private$.dims[3] = length(simulations)
  private$.arr = self$arr[,,simulations]
},
subset = function(rows,cols,mutate=TRUE){
  if('subset' %in% private$.debug){
    browser()
  }
  if(!mutate){
    rc = self$clone(TRUE)
    rc$subset(rows,cols,mutate=TRUE)
    return(rc)
  }

  if(missing(rows) && missing(cols)){
    return()
  }
  else if(missing(rows)){
    rows = 1:self$nrow
  }
  else if(missing(cols)){
    cols = 1:self$ncol
  }
  private$.arr = self$arr[rows,cols,,drop=FALSE]
  private$.dims = c(nrow(self$arr),ncol(self$arr),self$nsim)
  private$.dnames = dimnames(self$arr)
  if(length(self$rowData)>0){
    if(length(self$colData) > 0){
      self$dimData = list(
        lapply(self$rowData,function(x){x[rows,drop=FALSE]}),
        lapply(self$colData,function(x){x[cols,drop=FALSE]})
      )
    } else {
      self$rowData <- lapply(self$rowData,function(x){x[rows,drop=FALSE]})
    }
  } else if(length(self$colData)>0){
    self$colData <- lapply(self$colData,function(x){x[cols,drop=FALSE]})
  }
},
head = function(k,direction=2,mutate=FALSE){

  if('head' %in% private$.debug){
    browser()
  }
  if(k>dim(self$arr)[[direction]]){
    stop("The size of the head is too large.")
  }
  indices = 1:k
```

```
  if(direction==1){
    private$.arr = self$arr[indices,,,drop=FALSE]
    if(length(self$rowData)>0){
      private$.dimData[[1]] =
        lapply(self$rowData,function(x){x[indices,drop=FALSE]})
    }
  }
  else if(direction==2){
    private$.arr = self$arr[,indices,,drop=FALSE]
    if(length(self$colData)>0){
      private$.dimData[[2]] =
        lapply(self$colData,function(x){x[indices,drop=FALSE]})
    }
  }
  else{
    stop("This direction is not allowed.")
  }
  private$.dims = dim(self$arr)
  private$.dnames = dimnames(self$arr)
},
tail = function(k,direction=2){

  if('tail' %in% private$.debug){
    browser()
  }
  if(k>dim(self$arr)[[direction]]){
    stop("The size of the tail is too large.")
  }
  indices = (dim(self$arr)[[direction]]-k+1):dim(self$arr)[[direction]]

  if(direction==1){
    private$.arr = self$arr[indices,,,drop=FALSE]
    private$.dims = dim(self$arr)
    if(length(self$rowData)>0){
      self$rowData = lapply(self$rowData,function(x){x[indices,drop=FALSE]})
    }
  }
  else if(direction==2){
    private$.arr = self$arr[,indices,,drop=FALSE]
    private$.dims = dim(self$arr)
    if(length(self$colData)>0){
      self$colData = lapply(self$colData,function(x){x[indices,drop=FALSE]})
    }
  }
  else{
    stop("This direction is not allowed.")
  }
  private$.dims = dim(self$arr)
  private$.ndim = length(self$dims)
  private$.dnames = dimnames(self$arr)
},
lag = function(indices,mutate = TRUE,na.rm=FALSE){
```

```
if('lag' %in% private$.debug){
  browser()
}
if(mutate==FALSE){
  tmp = self$clone(TRUE)
  tmp$lag(indices=indices,mutate=TRUE)
  return(tmp)
}
if((1+max(indices)) > self$ncol){
  stop("We cannot go further back than the start of the matrix")
}
numLags = length(indices)
if(is.null(rownames(self$arr))){
  rownames(private$.arr) = 1:(dim(self$arr)[[1]])
}
rownames = replicate(numLags,rownames(self$arr))
colnames = colnames(self$arr)
private$.arr <- array(self$arr,c(dim(self$arr),numLags))
if(numLags <= 0){
  stop("indices must be nonempty for the calculation of lags to make sense.")
}
for(lag in 1:numLags){
  private$.arr[,(1+indices[[lag]]):self$ncol,,lag] <-
    self$arr[,1:(self$ncol-indices[[lag]]),,lag]
  if(indices[[lag]] > 0){
    private$.arr[,1:(indices[[lag]]),,lag] = NA
  }
}

private$.arr = aperm(self$arr,c(1,4,2,3))
private$.arr = array(self$arr,c(self$nrow*numLags,self$ncol,self$nsim))

lagnames = t(replicate(self$nrow,paste('L',indices,sep='')))

rownames(private$.arr) <-
  as.character(
    array(paste(lagnames,"R",rownames,sep=''),numLags*self$nrow)
  )
colnames(private$.arr) <- colnames

private$.dims[[1]] = self$nrow * numLags
if(!is.null(dimnames(self$arr))){
  private$.dnames = dimnames(self$arr)
}
if(length(self$rowData) > 0){
  self$rowData <- lapply(
    self$rowData,
    function(x){
      c(unlist(recursive=FALSE,lapply(1:numLags,function(y){x})))
    }
  )
}
if(na.rm==TRUE){
```

```
    self$subset(cols=!apply(self$arr,2,function(x){any(is.na(x))}))
  }
},
addRows = function(rows){
  if('addRows' %in% private$.debug){
    browser()
  }
  if(rows == 0){
    return()
  }
  abind(self$arr,array(NA,c(rows,self$ncol,self$nsim)),along=1) ->
    private$.arr
  private$.dims[[1]] = nrow(self$arr)
  private$.dnames = dimnames(self$arr)
  if(length(self$rowData) > 0){
    self$rowData = lapply(self$rowData,function(x){c(x,replicate(rows,NA))})
  }
},
addColumns = function(columns){
  "This function adds columns to the data."
  "@param columns The number of columns to add."
  if('addColumns' %in% private$.debug){
    browser()
  }

  if(columns == 0){
    return()
  }
  abind(private$.arr,array(NA,c(self$nrow,columns,self$nsim)),along=2) ->
    private$.arr
  private$.dims[2]= ncol(self$arr)
  private$.dnames = dimnames(private$.arr)
  if(length(self$colData) > 0){
    self$colData = lapply(self$colData,function(x){c(x,replicate(columns,NA))})
  }
},
scale = function(f,mutate=TRUE){
  if('scale' %in% private$.debug){
    browser()
  }
  if(!mutate){
    tmp = self$clone(TRUE)
    tmp$scale(f=f,mutate=TRUE)
    return(tmp)
  }
  private$.arr[] = f(private$.arr[])
},
diff = function(lag = 1,mutate=TRUE){
  if('diff' %in% private$.debug){
    browser()
  }
  if(!mutate){
    tmp = self$clone(TRUE)
```

```
    tmp$diff(lag=lag,mutate=TRUE)
    return(tmp)
  }
  if(lag == 0){
    if(!is.null(rownames(private$.arr))){
      rownames(private$.arr) = paste("D",lag,"R",rownames(private$.arr),sep='')
    }
    private$.dnames = dimnames(private$.arr)
    return()
  }
  if(lag < 0){
    stop("lag must be non-negative.")
  }
  rn = rownames(private$.arr)
  private$.arr <-
    self$simulations- self$lag(indices=lag,mutate=FALSE)$simulations
  if(!is.null(rn)){
    rownames(private$.arr) = paste("D",lag,"R",rownames(private$.arr),sep='')
  }
  private$.dnames = dimnames(private$.arr)
},
mutate = function(rows,cols,data){
  if('mutate' %in% private$.debug){
    browser()
  }
  tmpdata = data
  tmpdata = array(data,self$dims)
  data = as.array(data)

  if(missing(rows)){
    rows = 1:self$nrow
    if(!(is.null(self$cnames) || is.null(colnames(data)))){
      private$.dnames[[2]][cols] = colnames(data)
      colnames(private$.arr) = self$cnames
    }
  }
  if(missing(cols)){
    cols = 1:self$ncol
    if(!(is.null(self$rnames) || is.null(rownames(data)))){
      private$.dnames[[1]][rows] = rownames(data)
      rownames(private$.arr) = self$rnames
    }
  }
  if(is.null(dim(data))){
    stop("Not yet implemented for non-matrixlike objects")
  }
  if(length(dim(data)) > 3){
    stop("There are too many dimensions in data.")
  }
  if(length(dim(data)) == 3){
    if(dim(data)[[3]] == self$nsim){
      private$.arr[rows,cols,] = data
    } else if(dim(data)[[3]] == 1){
```

```
        private$.arr[rows,cols,] = replicate(self$nsim,data)
      }
    }
    else{
      private$.arr[rows,cols,] = replicate(self$nsim,data)
    }
  },
  summarize = function(FUNC,...){
    if('apply' %in% private$.debug){
      browser()
    }
    return(IncidenceMatrix$new(
      data=apply(private$.arr,c(1,2),FUNC),
      rowData=self$rowData,
      colData=self$colData,
      metaData=self$metaData)
    )
  }
),
active = list(
  sample = function(value){
    "Randomly extract a simulation"
    if("sample" %in% private$.debug){
      browser()
    }
    return = FALSE
    if(missing(value)){
      if(self$nsim < 1){
        return(private$.arr)
      }
      value = sample(self$nsim,1)
      return = TRUE
    }
    if(floor(value) != value){
      stop("sample must be an integer.")
    }

    private$.sample = value

    if(return){
      return(IncidenceMatrix$new(self))
    }
  },
  mat = function(value){
    private$.mat = adrop(private$.arr[,,private$.sample,drop=FALSE],3)
    rownames(private$.mat) = rownames(private$.arr)
    colnames(private$.mat) = colnames(private$.arr)
    return(private$.mat)
  },
  simulations = function(value){
    if(missing(value)){
      return(private$.arr)
    }
```

```
      stop("Do not write directly to the simulations")
    }
  )
)
```

ArrayData *ArrayData*

Description

An abstract class for storing data in the form of an n- dimensional array. It enforces core functionality that we want the data we model to have. This class covers any data which has two major axis, but has more minor axis.

Fields

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The size of the array.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : MatrixData

Is inherited by : AbstractIncidenceArray, AbstractSimulatedIncidenceMatrix, FrameData

Examples

```
SampleArrayData <- R6Class(
  inherit = ArrayData,
  public = list(
    initialize = function(data){
      if('array' %in% class(data)){
        private$.arr = data
        private$.dims = dim(data)
        private$.ndim = length(private$.dim)
        private$.dnames = dimnames(data)
      }
    }
  )
)
```

---

bomregions *sample dataset for use in vignettes*

Description

This is a dataset formerly part of the DAAG. We reproduce here for use in a vignette, since that package is no longer available on CRAN.

Author(s)

John Maindonald and W. John Braun

References

https://cran.r-project.org/src/contrib/Archive/DAAG/DAAG_0.5.8.tar.gz

---

DataContainer *DataContainer*

Description

A class for storing a matrix and relevant metaData. This class is the starting point for all of the Data oriented classes. However, its not really useful on its own, and serves mostly as a place-holder generic class in case we want to implement a drastically different type of model. This class should only need to be extended in the case where we implement something outside the scope of MatrixDataContainer

Fields

metaData Any data not part of the main data structure.

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Is inherited by : MatrixData

Examples

```
SampleDataContainer <- R6Class(
  inherit= DataContainer,
  public = list(
    initialize = function(...){
      self$metaData = list(...)
    }
  )
)
data <- SampleDataContainer$new(letters[1:10],1:10)
data
data$metaData
```

Forecast *Forecast*

Description

An abstract class for storing the results of forecasting. These classes do not contain any data directly, but instead contain a data object. Extend this class when you want to store the results of a model, and none of the convenience classes are applicable.

Fields

data The data used to create the forecast.

forecastMadeTime When the forecast was created.

forecastTimes The times the forecast is about.

model The model used to create the forecast.

Methods

binDist(cutoffs,include.lowest = FALSE,right = TRUE): This **must** be extended. Get the distribution of simulations of the data within fixed bins.

cutoffs	-	A numeric vector with elements to use as the dividing values for the bins.
include.lowest	-	logical, indicating if an x[i] equal to the lowest (or highest, for right = FALSE) breaks value should be inc
right	-	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.

Arguments

Value an ArrayData.

debug(string): A function for debugging the methods of this class. It calls the browser command.

In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

mean(): This **must** be extended. This method extracts the elementwise mean of the forecast. This function will not change the number of rows or columns in the data, but will convert probabilistic estimates into deterministic ones.

Arguments

Value a MatrixData.

median(): This **must** be extended. This method extracts the elementwise median of the forecast. This function will not change the number of rows or columns in the data, but will convert probabilistic estimates into deterministic ones.

Arguments

Value a MatrixData.

quantile(alphas,na.rm=FALSE): This **must** be extended. Get the cutoffs for each percentile in alphas.

alphas - A numeric vector with elements between `0` and `1` of percentiles to find cutoffs for.
na.rm - A boolean regarding whether to remove NA values before computing the quantiles.

Arguments

Value an ArrayData.

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Is inherited by : SimpleForecast, SimulatedForecast

Examples

```
SimpleForecast <- R6Class(
  classname = "SimpleForecast",
  inherit = Forecast,
  private = list(
    .data = MatrixData$new()
  ),
  public = list(
    binDist = function(cutoffs){
      stop("This doesn't really make sense.")
    },
    mean = function(){
      return(self$data)
    },
    median = function(){
      return(self$data)
    },
    quantile = function(alphas){
      stop("This doesn't really make sense.")
    },
    initialize = function(data,forecastTimes){
     if(missing(data) && missing(forecastTimes)){
     return()
     }
      if(data$ncol != length(forecastTimes)){
        stop("The number of columns should be the number of times forecasted.")
      }
      private$.forecastMadeTime = now()
      private$.forecastTimes = forecastTimes
      private$.data = data
    }
  ),
  active = list(
  )
)
```

ForecastModel *ForecastModel*

Description

A model for predicting multiple time steps into the future. You should extend this class if you are predicting over some future time period and none of the convenience classes which extend forecast are applicable.

Methods

forecast(newdata,steps): This method **must** be extended. This function is similar to predict, in that it predicts the rows of the input data, however it can predict multiple timesteps into the future, instead of a single timestep.

newdata	-	The data to forecast from.
steps	-	The number of timesteps into the future to predict.

Arguments
Value A forecast

`FrameData` *FrameData*

Description

An abstract class for storing data in the form of a data frame. It enforces core functionality that we want the data we model to have. This class covers any data which is stored as list.

Fields

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The size of the array.

frame The data frame this class is responsible for.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : ArrayData

Is inherited by : AbstractObservationList, RelationalData

Examples

```
SampleFrameData <- R6Class(
  inherit = FrameData,
  public = list(
    initialize = function(data){
      private$.frame = data
      private$.arr = data
      private$.mat = data
      private$.dims = dim(data)
      private$.ndim = length(private$.dims)
      private$.dnames = dimnames(data)
    }
  )
)
```

`IncidenceForecast` *IncidenceForecast*

Description

A basic concrete SimulatedForecast class.

Fields

data The prediction this model is responsible for. The data should be of class SimulatedIncidenceMatrix

forecastMadeTime When the forecast was created.

forecastTimes The times the forecast is about.

model The model used to create the forecast.

nsim The number of simulations.

sample Draw a random sample from the possible model predictions. Please see implementation of the data for the properties of the sampling.

Methods

binDist(cutoffs,include.lowest = FALSE,right = TRUE): Get the distribution of simulations of the data within fixed bins.

cutoffs	-	A numeric vector with elements to use as the dividing values for the bins. -Inf, and Inf will be added auto
include.lowest	-	logical, indicating if an x[i] equal to the lowest (or highest, for right = FALSE) breaks value should be inc
right	-	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.

Arguments

Value an ArrayData.

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string	-	The name(s) of methods to debug as a character vector

Arguments

initialize(data=SimulatedIncidenceMatrix$new(),forecastTimes=c()): Create a new IncidenceForecast.

data	-	The data to initialize with

Arguments

mean(trim = 0,na.rm = FALSE): This method extracts the elementwise mean of the forecast. This function will not change the number of rows or columns in the data, but will convert probabilistic estimates into deterministic ones.

trim - the fraction (0 to 0.5) of observations to be trimmed from each end of 'x' before the mean is computed. Values of
na.rm - a logical value indicating whether 'NA' values should be stripped before the computation proceeds.

Arguments

Value An IncidenceMatrix with the mean over all simulations.

median(na.rm=FALSE): This method extracts the elementwise median of the forecast. This function will not change the number of rows or columns in the data, but will convert probabilistic estimates into deterministic ones.

na.rm - a logical value indicating whether 'NA' values should be stripped before the computation proceeds.

Arguments

Value a MatrixData.

quantile(probs,na.rm=FALSE,names=TRUE,type=7): Get the cutoffs for each percentile in probs.

probs - A numeric vector with elements between `0` and `1` of percentiles to find cutoffs for. (Values up to '2e-14' outside t
na.rm - logical; if true, any 'NA' and 'NaN"s are removed from 'x' before the quantiles are computed.
names - logical; if true, the result has a 'names' attribute. Set to 'FALSE' for speedup with many 'probs'.
type - an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.

Arguments

Value an ArrayData where the rows and columns correspond to the .

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : `SimulatedForecast`

Examples

```
data = SimulatedIncidenceMatrix$new(
  IncidenceMatrix$new(matrix(1:9,3,3)),
  nsim=3
)
data$addError(cols = 3,type='Poisson')
forecast = IncidenceForecast$new(data,forecastTimes=c(FALSE,FALSE,TRUE))
forecast
forecast$forecastTimes
forecast$forecastMadeTime
forecast$data$mat
forecast$nsim
forecast$sample$mat
forecast$sample$mat
forecast$mean()$mat
forecast$median()$mat
forecast$binDist(1:4*4)$arr
forecast$quantile(c(.05,.5,.95))$arr
```

`IncidenceMatrix` *IncidenceMatrix*

Description

A class for storing a matrix and relevant metaData.

Fields

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

frame A data.frame representation of the IncidenceMatrix. The frame contains all rows/columns as rows of the data frame, and the rows, columns, values and all metadata types as columns. This is similar to the melt function.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the columns of the data.

Methods

addColumns(columns): This function adds empty columns to the right side of the data.

columns - The number of columns to add.

Arguments

addRows(rows): This function adds empty rows to the data.

rows - The number of rows to add.

Arguments

as_sts(pop_name = 'pop',freq_name = 'freq',start_name = 'start',epoch_name = 'epoch',map_name = 'map',neighbourhood_name = 'neighbourhood'): Convert this object to an sts object with as much information as possible

pop_name - string with the name of rowData to use as the population of the sts object.
freq_name - string with the name of metaData to use as the frequency of the sts object.
start_name - string with the name of metaData to use as the start of the sts object.
epoch_name - string with the name of the colData to use as the epoch of the sts object.
map_name - string with the name of the rowData to use as the map of the sts object.
neighbourhood_name - string with the name of the rowData to use as the neighbourhood of th ests object.

Arguments

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

diff(lag = 1,mutate=TRUE): This function replaces the matrix value at column i with the difference. between the values at columns i and (i-lag).

lag - How far back to diff. Defaults to 1.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

head(k,direction=2,mutate=TRUE): Select the last `k` slices of the data in dimension `direction`.

k - The number of slices to keep.
direction - The dimension to take a subset of. 1 for row, 2 for column.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method w

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

initialize(data=matrix(),metaData=list(),rowData=list(),colData=list(),cellData = list()): Create a new `IncidenceMatrix` object from its components.

data - Either a matrix, or an object of class MatrixData. If data is a matrix, it will form the mat field. If data is a MatrixDa

Arguments

lag(indices,mutate=TRUE,na.rm=FALSE): This function replaces the current matrix with a new matrix with one column for every column, and a row for every row/index combination. The column corresponding to the row and index will have the value of the original matrix in the same row, but index columns previous. This shift will introduce NAs where it passes off the end of the matrix.

indices - A sequence of lags to use as part of the data. Note that unless this list contains 0, the data will all be shifted back
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will
na.rm - Whether to remove NA values generated by walking off the edge of the matrix.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

lead(indices,mutate=TRUE,na.rm=FALSE): This function replaces the current matrix with a new matrix with one column for every column, and a row for every row/index combination. The column corresponding to the row and index will have the value of the original matrix in the same row, but index columns ahead.

indices - A sequence of leads to use as part of the data. Note that unless this list contains `0`, the data will all be shifted fore
mutate - Whether to modify this object, or create and return a modified object.
na.rm - Whether to remove the NA columns that result where the lead goes off the edge of self$simulations.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

mutate(rows,cols,data): This function is a way to modify the data as though it were a matrix.

rows - Which rows to modify. These can be numeric or names.
cols - Which cols to modify. These can be numeric or names.
data - The data to change the chosen values to. It needs to be the right shape.

Arguments

scale(f,mutate=TRUE): This function rescales each element of our object according to f

f - a function which takes in a number and outputs a rescaled version of that number
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

subset(rows,cols,mutate=TRUE): Select the data corresponding to the rows `rows` and the columns `columns`. `rows` and `columns` can be either numeric or named indices.

rows - An row index or list of row indices which can be either numeric or named.
cols - An column index or list of column indices which can be either numeric or named.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method will

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

tail(k,direction=2,mutate=TRUE): Select the last k slices of the data in dimension `direction`.

k - The number of slices to keep.
direction - The dimension to take a subset of. 1 for row, 2 for column.
mutate - Whether to change the original instance, or create a new one. If FALSE, the instance performing the method w

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : AbstractIncidenceMatrix

Examples

```
data = IncidenceMatrix$new(matrix(1:9,3,3))
data$mat
data$nrow
data$ncol
data$colData = list(1:3,letters[1:3])
data$colData
data$addColumns(2)
data$colData
data$mat
data$diff(1)
data$mat
data$lag(1)
data$mat
data$head(1,1)
data$mat
data$tail(2,2)
data$mat
data$mutate(data=3)
data$mat
data = IncidenceMatrix$new(matrix(1:9,3,3))
data$scale(function(x){x^2},mutate=FALSE)$mat
data$mat
data$subset(rows=1,cols=2)
data$mat
```

MatrixData *MatrixData*

Description

An abstract class for storing metaData relevant to a matrix. This class forms the backbone of the data structures. It enforces the core functionality we want the data we model to have. This class covers any data which has two major axes, and can be thought of as a matrix. It also can cover the case where the data is matrix-like, but our representation of the data is not matrix-like. ## TODO: make an example for this (think cholera data).

Fields

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the columns of the data.

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : DataContainer

Is inherited by : AbstractIncidenceMatrix, ArrayData

Examples

```
SampleMatrixData <- R6Class(
  inherit = MatrixData,
  public = list(
    initialize = function(data){
      if('matrix' %in% class(data)){
        private$.mat = data
        private$.nrow = nrow(data)
        private$.ncol = ncol(data)
```

```
          private$.rnames = rownames(data)
          private$.cnames = colnames(data)
        }
      }
    )
)
```

`Model` *Model*

Description

An abstract class for making predictions. This class contains only what's necessary for data based modeling. For the most part, the other model classes in this package are there to make extending this class easier. You should extend this class directly if you are making a model which does not forecast, and none of the convenience classes seem appropriate.

Methods

fit(data): This method **must** be extended. Get the model ready to predict.

data - The data to fit the model to.

Arguments

predict(newdata): This method **must** be extended. Predict using the model

newdata - the data to predict.

Arguments

`MoveAheadModel` *MoveAheadModel*

Description

An autoregressive model which assumes everything stays the same.

Fields

data A MatrixData containing the data used in fitting of the model.

maxPredCol The farthest back column from the output value used in prediction.

nsim The number of stochastic simulations to perform.

predCols An array of which columns are used in prediction. 1 means the column before the prediction, 2 the one before that etc.

stochastic A string with the family of stochastic noise to apply to the recursive process. Currently only accepts 'Deterministic' and 'Poisson' as values.

Methods

fit(data): Fit the model for predicting. This method breaks down the fitting process into two parts, data preparation, and the model fitting.

data - The data to use to fit the model. This object should be a MatrixData object.

Arguments

fit_(): This method is included for compliance with the standard but does nothing. This model does not need to be fit.

Arguments

forecast(newdata = private$.data,steps=1): Using a model previously fit with `fit` to predict the next `steps` columns. This function assumes that all of the data preprocessing has already been taken care of. This function is similar to predict, except that it can predict multiple time steps into the future instead of a single timestep.

newdata - The data to forecast from.
steps - The number of timesteps into the future to predict.

Arguments

Value private$output This function should both modify and return private$output.

initialize(nsim = 3): Create a new MoveAheadModel

nsim - The number of stochastic simulations to use.

Arguments

predict(newdata): Use a model previously fit with fit to predict. This function does not assume any data preprocessing. This function should not, in general need to be overwritten by the user. The main prediction function is predict_, so please modify that function instead if possible.

newdata - Optional. The data used to predict. This data should be a MatrixData object.

Arguments

predictRow(newdata,row,col): Using a model previously fit with `fitRow` to predict the `row`th row of the next column. This function does not assume any data preprocessing. Since the predict method predicts every row at the the same time, we include this method for predicting only a single row.

newdata - Optional. The data use to predict. This can either be a matrix, appropriately formatted lag vector, or NULL to p
row - The row to predict the value of.
col - This is for internal use only.

Arguments

predictRow_(row,col=0): This method predicts a particular row.

row - The row to predict
col - This is for internal use.

Arguments

predict_(col=0): This method predicts as much as possible about the next time step.

col - This is for internal use.

Arguments

prepareFitData(data): This method gets data ready for fitting and stores it in the model for later use.

data - The data used to fit the model.

Arguments

prepareForecastData(data): This method gets data ready to use to forecast and stores it in the model for later use.

data - The data to use when forecasting.

Arguments

prepareOutputData(inputData,steps=0): This method prepares the output for predict or forecast.

inputData - The data being used to predict/forecast.
steps - The number of steps in the forecast. (0 for predict).

Arguments

preparePredictData(newdata): This method gets data ready to use to predict and stores it in the model for later use.

newdata - The data to use when fitting the model.

Arguments

See Also

Inherits from : RecursiveForecastModel

Examples

```
MoveAheadModel <- R6Class(
  classname = 'MoveAheadModel',
  inherit = RecursiveForecastModel,
  private = list(
    .data = IncidenceMatrix$new(),
    newdata = IncidenceMatrix$new(),
    output = ArrayData$new(),
   .nsim = 3,
    .predCols = c(as.integer(1)),
    .maxPredCol = as.integer(1)
  ),
  public = list(
    initialize = function(nsim = 3){
      private$.nsim = nsim
    },
    fit_ = function(){
      if(self$data$ncol <= self$predCols){
        stop("We cannot go further back than the start of the matrix.")
      }
    },
    predictRow_ = function(row,col=0){
      predict_(col)
    },
    predict_ = function(col=0){
      if('predict_' %in% private$.debug){
        browser()
      }
      if(col == 0){
        col=1:private$output$ncol
      }
      private$output$mutate(
```

```
          cols=col,
          data = SimulatedIncidenceMatrix$new(
            private$newdata,
            private$.nsim
          )$simulations
        )
      },
      prepareFitData = function(data){
        private$.data = data$clone(TRUE)
      },
      preparePredictData = function(newdata){
        if('preparePredictData' %in% private$.debug){
          browser()
        }
        private$newdata = SimulatedIncidenceMatrix$new(data=newdata,nsim=self$nsim)
        private$.nrow = private$newdata$nrow
        private$newdata$addColumns(min(self$predCols))
        private$newdata$lag(min(self$predCols),na.rm=FALSE)
        private$newdata$subset(cols=2:private$newdata$ncol)
      },
      prepareForecastData = function(data){
        if(self$data$ncol <= self$predCols){
          stop("We cannot go further back than the start of the matrix.")
        }
        private$newdata = data$clone(TRUE)
        private$.nrow = private$newdata$nrow
        private$newdata$addColumns(min(self$predCols))
        private$newdata$lag(min(self$predCols),na.rm=TRUE)
      },
      prepareOutputData = function(inputData,steps=0){
        private$output = SimulatedIncidenceMatrix$new(inputData,private$.nsim)
        if(steps > 0){
          private$output$addColumns(steps)
        }
      }
    ),
    active = list(
      nsim = function(value){
        private$defaultActive(type='private',name='.nsim',val=value)
      }
    )
  )
```

ObservationList *ObservationList*

Description

ObservationList is a class for recording instances of observations. If a matrix is wide form, these observations are in long form.

Fields

aCellData Variable which stores the column names of `self$frame` associated with each dimension of `self$arr`, but not defining them.

aDimData Variable which stores the column names of `self$frame` associated with each dimension of `self$arr`, but not defining them.

aDims Variable which stores the column names of `self$frame` defining each dimension of `self$arr`

aVal Variable which stores the column names of `self$frame` defining the values of self$arr

aggregate A function used to aggregate elements of `self$frame` when it is grouped. This function should take a single table as input, and summarize each relevant column when grouped.

arr An array of aggregate data pulled from the frame. See `formArray` for details

cellData A list of data associated to the cells of `self$arr`

cnames The names of the rows of `self$arr`

colData A list of data associated to the columns of `self$arr`

dimData A list of data associated with each dimension of `arr`

dims The dimensions of `arr`

dnames The names of the slices of each dimension of `self$arr`

frame The data frame this object is responsible for.

mat A matrix pulled from a cross section of `self$arr`

ncol The number of columns in `self$arr`

ndim The number of dimensions of `arr`

nrow The number of rows in `self$arr`

rnames The names of the rows of `self$arr`

rowData A list of data associated to the rows of `self$arr`

slice Which slice of `self$arr` to look at for `self$mat`

Methods

formArray(...,val,dimData=list(),metaData=list(),cellData = list()): In order to use an ObservationList as an ArrayData, you need to select which columns to use to form the dimensions of the array. Optionally, you can also assign some of the columns to be associated with each dimension (or cell). Note that `aggregate` is used to determine how to deal with multiple frame associated with a particular grouping.

...	-	Column names of columns which, in order should form the dimensions of the array
val	-	The attribute of frame to use for the values of the array (must `aggregate_` to a numeric type)
dimData	-	A list containing for each dimension of the array, the attribute(s) of `frame` which are associated with that dimer

Arguments

initialize(data=tibble::tibble(),...): Create a new ObservationList with `frame` given by `data`

data - A data frame to use as the frame of the ObservationList
... - A list of arguments to pass to the formArray function. These arguments determine how the ObservationList behaves

Arguments

`RecursiveForecastModel`
RecursiveForecastModel

Description

A model for recursively predicting multiple timesteps into the future. This class implements a particular type of forecasting, where a predict method is called to predict each successive time step in order. Extend this class if you have a predict method, but not a special forecast method.

Fields

data A MatrixData containing the data used in fitting of the model.

maxPredCol The farthest back column from the output value used in prediction.

predCols An array of which columns are used in prediction. 1 means the column before the prediction, 2 the one before that etc.

stochastic A string with the family of stochastic noise to apply to the recursive process. Currently only accepts 'Deterministic' and 'Poisson' as values.

Methods

fit(data): Fit the model for predicting. This method breaks down the fitting process into two parts, data preparation, and the model fitting.

data - The data to use to fit the model. This object should be a MatrixData object.

Arguments

fit_(): This method **must** be extended. Fit the model for predicting all of the rows. Assumes the data has been put into place.

Arguments

forecast(newdata = private$.data,steps=1): Using a model previously fit with `fit` to predict the next `steps` columns. This function assumes that all of the data preprocessing has already been taken care of. This function is similar to predict, except that it can predict multiple time steps into the future instead of a single timestep.

newdata - The data to forecast from.
steps - The number of timesteps into the future to predict.

Arguments

Value private$output This function should both modify and return private$output.

predict(newdata): Use a model previously fit with fit to predict. This function does not assume any data preprocessing. This function should not, in general need to be overwritten by the user. The main prediction function is predict_, so please modify that function instead if possible.

newdata - Optional. The data used to predict. This data should be a MatrixData object.

Arguments

predictRow(newdata,row,col): Using a model previously fit with `fitRow` to predict the `row`th row of the next column. This function does not assume any data preprocessing. Since the predict method predicts every row at the the same time, we include this method for predicting only a single row.

newdata - Optional. The data use to predict. This can either be a matrix, appropriately formatted lag vector, or NULL to p
row - The row to predict the value of.
col - This is for internal use only.

Arguments

predictRow_(row): This method **must** be extended. Using a model previously fit with `fitRow` to predict the `row`th row of the next column. This function assumes that all of the data preprocessing has already been taken care of.

row - The row to predict the value of.

Arguments

Value private$output The return value should be both stored in private$output and returned using return. This should contain the results of the prediction in a Forecast object with dimensions similar to private$newdata.

predict_(col=0): This method **must** be extended. Using a model previously fit with `fit` to predict each row of the next column. This function assumes that all of the data preprocessing has already been taken care of.

col - Which columns of private$output should be modified. This parameter is mainly used in forecast, but could be used to

Arguments

Value private$output The return value should be both stored in private$output and returned using return. This should contain the results of the prediction in a Forecast object with dimensions similar to private$newdata.

prepareFitData(data): This method **must** be extended. Take `data` and store it in the object. This allows the data to be referenced later in predict() where newdata is NULL. It may also be helpful to put other data preparation steps in this method, so that the fit function runs more smoothly.

data - The data to prepare.

Arguments

prepareForecastData(data): This method **must** be extended. This function takes input data and prepares it to forecast. It should in principle, be similar to preparePredictData, but sometimes forecasting requires different preparation from prediction.

data - The data to prepare for forecasting from.

Arguments

Value private$newdata Store the processed value here, so that forecast can access it.

prepareOutputData(inputData,steps=0): This method **must** be extended. This function takes the input data and constructs an appropriate container for the output of the model.

inputData - The input to the model. Used to determine properties of the container.
steps - If the input data is used as part of a forecast, the number of steps is passed in case the output size depends on t

Arguments

preparePredictData(newdata): This method **must** be extended. Take `newdata` and use it to prepare the model, so that predicting doesn't need to directly reference it. This allows the model to make multiple predict_, predictRow_, or forecast calls without re-allocating the data every time.

newdata - The data to prepare.

Arguments

See Also

Is inherited by : `MoveAheadModel`

`RelationalData` *RelationalData*

Description

An abstract class for storing data in the form of a data frame. It enforces core functionality that we want the data we model to have. This class covers any data which is stored as list.

Fields

arr This is the full array. For extensibility, it cannot be written to directly and must be modified through methods.

cellData A list of metadata associated with the cells of the data.

cnames The names of columns in the data.

colData A list of metadata associated with the columns of the data.

dimData The data associated with each dimension of the array.

dims The size of the array.

dnames The size of the array.

frame The data frame this class is responsible for.

mat This is the matrix. For extensibility, it cannot be written to directly and must be modified through methods.

metaData Any data not part of the main data structure.

ncol The number of columns in the data.

ndim The number of dimensions of the array.

nrow The number of rows in the data

rnames The names of rows in the data.

rowData A list of metadata associated with the rows of the data.

tables The tables which make up the relational database.

Methods

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

... - This function should take in any arguments just in case.

Arguments

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : FrameData

Is inherited by : AbstractRelationalTables

Examples

```
library(dplyr)
library(reshape2)
SampleRelationalTables <- R6Class(
  inherit = AbstractRelationalTables,
  public = list(
    initialize = function(...){
      private$.tables = list(...)
      if(!all(sapply(private$.tables,function(x){is.data.frame(x)}))){
        stop("All arguments must be data frames")
      }
    },
    updateFrame = function(){
      private$.frame = Reduce(x = private$.tables,f = left_join)
    },
    updateArray = function(){
      val <- names(self$frame)[1]
      dims <- names(self$frame[2:ncol(self$frame)])
      private$.arr <- self$frame %>%
        group_by_(.dots=setNames(dims,NULL)) %>%
        summarize_all(sum) %>%
        ungroup() %>%
        acast(as.formula(paste(dims,collapse='~')),value.var=val)
      mode(private$.arr) = 'numeric'
      private$.dims = dim(private$.arr)
      private$.nrow = private$.dims[1]
      private$.ncol = private$.dims[2]
      private$.ndim = length(private$.dims)
      private$.dnames = dimnames(private$.arr)
    }
  ),
  active = list(
```

```
    mat = function(value){
      if(self$ndim <= 2){
        return(self$arr)
      }
    return(extract(private$.arr,indices = rep(self$ndim-2,x=1),dims = 3:self$ndim,drop=TRUE))
    }
  )
)
```

`SimpleForecast` *SimpleForecast*

Description

A basic concrete SimulatedForecast class.

Fields

data The data used to create the forecast.

forecastMadeTime When the forecast was created.

forecastTimes The times the forecast is about.

model The model used to create the forecast.

Methods

binDist(cutoffs): This throws an error. This method is not meaningful for this data.

cutoffs - A numeric vector with elements to use as the dividing values for the bins.

Arguments

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string - The name(s) of methods to debug as a character vector

Arguments

initialize(data,forecastTimes): Create a new SimpleForecast.

data - The data to initialize with
forecastTimes - Boolean representing which times are forecasted, and which times are not.

Arguments

mean(): This method returns the data. It is included for compliance.

Arguments

Value a MatrixData.

median(): This method returns the data. It is included for compliance.

Arguments

Value a MatrixData.

quantile(alphas,na.rm=FALSE): This throws an error. This method is not meaningful for this data.

alphas - A numeric vector with elements between `0` and `1` of percentiles to find cutoffs for.
na.rm - A boolean regarding whether to remove NA values before computing the quantiles.

Arguments

Value an ArrayData.

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : `Forecast`

Examples

```
data = IncidenceMatrix$new(matrix(1:9,3,3))
forecast = SimpleForecast$new(data,forecastTimes=c(FALSE,FALSE,TRUE))
forecast
forecast$forecastTimes
forecast$forecastMadeTime
forecast$data$mat
forecast$nsim
forecast$mean()$mat
forecast$median()$mat
```

`SimulatedForecast` *SimulatedForecast*

Description

This class is a forecast where the data is many simulated trials.

Fields

data The data used to create the forecast.

forecastMadeTime When the forecast was created.

forecastTimes The times the forecast is about.

model The model used to create the forecast.

nsim The number of simulations.

sample Draw a random sample from the possible model predictions. Please see implementation of the data for the properties of the sampling.

Methods

binDist(cutoffs,include.lowest = FALSE,right = TRUE): Get the distribution of simulations of the data within fixed bins.

cutoffs	-	A numeric vector with elements to use as the dividing values for the bins. -Inf, and Inf will be added auto
include.lowest	-	logical, indicating if an x[i] equal to the lowest (or highest, for right = FALSE) breaks value should be inc
right	-	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.

Arguments

Value an ArrayData.

debug(string): A function for debugging the methods of this class. It calls the browser command. In order for methods to opt into to debugging, they need to implement the following code at the beginning: if(<method_name> %in% private$.debug){browser()}. This method exists, because the debugger is not always intuitive when it comes to debugging R6 methods.

string	-	The name(s) of methods to debug as a character vector

Arguments

initialize(...): This function **should** be extended. Create a new instance of this class.

...	-	This function should take in any arguments just in case.

Arguments

mean(trim = 0,na.rm = FALSE): This method extracts the elementwise mean of the forecast. This function will not change the number of rows or columns in the data, but will convert probabilistic estimates into deterministic ones.

trim - the fraction (0 to 0.5) of observations to be trimmed from each end of 'x' before the mean is computed. Values of
na.rm - a logical value indicating whether 'NA' values should be stripped before the computation proceeds.

Arguments

Value An IncidenceMatrix with the mean over all simulations.

median(na.rm=FALSE): This method extracts the elementwise median of the forecast. This function will not change the number of rows or columns in the data, but will convert probabilistic estimates into deterministic ones.

na.rm - a logical value indicating whether 'NA' values should be stripped before the computation proceeds.

Arguments

Value a MatrixData.

quantile(probs,na.rm=FALSE,names=TRUE,type=7): Get the cutoffs for each percentile in probs.

probs - A numeric vector with elements between `0` and `1` of percentiles to find cutoffs for. (Values up to '2e-14' outside t
na.rm - logical; if true, any 'NA' and 'NaN"s are removed from 'x' before the quantiles are computed.
names - logical; if true, the result has a 'names' attribute. Set to 'FALSE' for speedup with many 'probs'.
type - an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.

Arguments

Value an ArrayData where the rows and columns correspond to the .

undebug(string): A function for ceasing to debug methods. Normally a method will call the browser command every time it is run. This command will stop it from doing so.

string - The name(s) of the methods to stop debugging.

Arguments

See Also

Inherits from : `Forecast`

Is inherited by : `IncidenceForecast`

Examples

```
IncidenceForecast <- R6Class(
  classname = "IncidenceForecast",
  inherit = SimulatedForecast,
  private = list(
    .data = AbstractSimulatedIncidenceMatrix$new()
  ),
  public = list(
    initialize = function(data=SimulatedIncidenceMatrix$new(),forecastTimes=c()){
      if(data$ncol != length(forecastTimes)){
        stop("The number of columns should be the number of times forecasted.")
      }
      private$.forecastMadeTime = now()
      private$.forecastTimes = forecastTimes
      private$.data = data
    }
  ),
  active = list(
    data = function(value){
      private$defaultActive(".data","private",value)
    }
  )
)
```

`SimulatedIncidenceMatrix`

SimulatedIncidenceMatrix

Description

A class for storing lists of IncidenceMatrices

Fields

mat A matrix containing a single sample. By default, it is the first sample. See `self$sample` for how to change it.

sample Return a random sample from the simulations. Alternatively, select a sample to use with `self$mat` by assigning a value.

simulations The simulations this structure is responsible for. This is another name for `self$arr`.

Methods

addColumns(columns): This function adds columns to `self$simulations`

columns - The number of columns to add.

Arguments

addError(type,rows,cols,mutate = TRUE): Add error to the simulations according to a distribution.

type	-	The type of distribution as a string. Currently 'Poisson' is allowed.
rows	-	Which rows to affect.
cols	-	Which columns to affect.
mutate	-	Whether to modify this object, or create and return a modified object.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

addRows(rows): This function adds rows to `self$simulations`

rows	-	The number of rows to add.

Arguments

diff(lag = 1,mutate=TRUE): This function replaces the matrix value at column i with the differences between the values at column `i` and `i-lag`.

lag	-	How far back to diff.
mutate	-	Whether to modify this object, or create and return a modified object.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

head(k,direction=2,mutate=FALSE): Take the first `k` slices of `self$simulations`

k	-	How many slices to keep
direction	-	Which dimension to take a subset of. 1 is rows, 2 is columns, 3 is simulations
mutate	-	Whether to modify this object, or create and return a modified object.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

initialize(data=MatrixData$new(),nsim=1): Create a new SimulatedIncidenceMatrix.

data	-	The data to use for the simulation. Can be a list of IncidenceMatrices, or a single IncidenceMatrix.
nsim	-	The number of simulations. If `data` is a list, this should be the length of the list. If `data` is an IncidenceMatrix, this

Arguments

lag(indices,mutate = TRUE,na.rm=FALSE): This function replaces the current matrix with a new matrix with one column for every column, and a row for every row/index combination. The column corresponding to the row and index will have the value of the original matrix in the same row, but index columns previous.

indices - A sequence of lags to use as part of the data. Note that unless this list contains `0`, the data will all be shifted back
mutate - Whether to modify this object, or create and return a modified object.
na.rm - Whether to remove the NA columns that result where the lag goes off the edge of self$simulations.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

lead(indices,mutate = TRUE,na.rm=FALSE): This function replaces the current array with a new array with one column for every column, and a row for every row/index combination. The column corresponding to the row and index will have the value of the original array in the same row, but index columns ahead.

indices - A sequence of leads to use as part of the data. Note that unless this list contains `0`, the data will all be shifted bac
mutate - Whether to modify this object, or create and return a modified object.
na.rm - Whether to remove the NA columns that result where the lead goes off the edge of self$simulations.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

mean(): Compute the mean over all simulations.

Arguments

Value An IncidenceMatrix where return$mat is the elementwise mean of self$arr

median(): Compute the median over all simulations.

Arguments

Value An IncidenceMatrix where return$mat is the elementwise median of self$arr

mutate(rows,cols,sims,data): This function changes the information stored in `self$simulations`

rows - The rows to change.

cols - The columns to change.
sims - Which simulations to affect.
data - The data to change to. Can be either array-like or matrix-like. If its matrix-like it will overwrite all of the dimensio

Arguments

scale(f,mutate=TRUE): This function rescales each element of our object according to a function.

f - The function we rescale by. This function takes in a number and outputs a rescaled version of that number.
mutate - Whether to modify this object, or create and return a modified object.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

subsample(simulations,mutate=TRUE): Choose only some of the simulations.

simulations - Which simulations to keep
mutate - Whether to modify this object, or create and return a modified object.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

subset(rows,cols,mutate=TRUE): Take a subset of the object as though it were a matrix.

rows - Numeric, named, or logical denoting which rows to select
cols - Numeric, named, or logical denoting which columns to select
mutate - Whether to modify this object, or create and return a modified object.

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

summarize(FUNC,...): Apply a function to every simulation.

FUNC - The function to apply.
... - Any arguments to `FUNC` other than the matrix.

Arguments

Value An IncidenceMatrix where `return$mat[i,j]` is the same as `FUNC(self$arr[i,j,])`

tail(k,direction=2): Take the last `k` slices of `self$simulations`

k - How many slices to keep
direction - Which dimension to take a subset of. 1 is rows, 2 is columns, 3 is simulations

Arguments

Value If `mutate=FALSE`, a clone of this object will run the method and be returned. Otherwise, there is no return.

Index