Package 'expMRF'

March 27, 2014

Type Package

Title Markov Random Field for Exponential Family

Version 1.0

Date 2014-03-25

Author Ying-Wooi Wan, Zhandong Liu

Maintainer Ying-Wooi Wan <yingwoow@bcm.edu>

Deoends glmnet, igraph, huge

Description
This package contains methods to employ Markov Random Field (MRF) for exponential family.

License GPL2.0

LazyLoad yes

R topics documented:

`expMRF-package` *Markov random field (MRF, or Markov network) for exponential family.*

Description

This package includes multiple functions to implement the local Log-Linear Graphical Model based on pair-wise markov network using efficient, parallel algorithm. This algorithm employs neighborhood selection to infer network structure. Stability selection is used in selecting the optimal network.

Details

Package:	expMRF
Type:	Package
Version:	1.0
Date:	2014-03-25
License:	What license is it under?

~~ An overview of how to use the package, including the most important ~~ ~~ functions ~~

Author(s)

Ying-Wooi Wan, Zhandong Liu Maintainer: Ying-Wooi Wan <yingwoow@bcm.edu> ~~ The author and/or maintainer of the package ~~

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

See Also

~~ Optional links to other man pages, e.g. ~~ ~~ <pkg> ~~

Examples

```
library(expMRF)

n = 100
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace =T), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

# Run LPGM
lpgm.path.all.p = LPGM.select(t(simData), nlams=10, N=10, beta=0.05, nCpus=2, parallel=T)
lpgm.path.all.p

plot(lpgm.path.all.p, fn="lpgm.opt.net.pdf")
```

Bsublin *Sublinear function*

Description

Transform the value of a data matrix (X) by a sub-linear function

Usage

```
Bsublin(X, R, R0 = 0)
```

Arguments

X	a data matrix
R	upper-bound threshold value. Note: R should be great than 0
R0	lower-bound threshold value, default to 0

Details

Given two threshold values R and R0, s.t R > 0, R0 > 0 and R > R0

Each element x in X is transformed as follows:

```
x = x,                                          if x <= R0
 x = (-x^2+ 2*R*x - R0^2)/(2 * (R - R0)),       if R0 < x <= R
 x = (R + R0)/2,                                if x > R
```

Value

Bx	the transformed data matrix, of the same dimension as original data matrix X

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, R, R0 = 0)
{
    Bx = X
    Bx[X > R] = (R + R0)/2
    ind = X > R0 & X <= R
    Bx[ind] = (-X[ind]^2 + 2 * R * X[ind] - R0^2)/(2 * (R - R0))
    return(Bx)
  }
```

Copula.Norm.Pois *Copula transform a matrix from normal to Poisson*

Usage

```
Copula.Norm.Pois(X, lambda)
```

Arguments

X	a nxp data matrix of Gaussians
lambda	the Poisson mean for the transformation

Value

Y	a nxp Copula transformed data matrix

Examples

```
X <- matrix(rnorm(20), nrow=5, ncol=4)
transX <- Copula.Norm.Pois(X, lambda=1)

## The function is currently defined as
function (X, lambda)
{
    n = nrow(X)
    p = ncol(X)
    val = 0
    dcuts = NULL
    cnt = 0
    while (val < max(0.9999, 1 - 2/(n * p))) {
```

```
        val = ppois(cnt, lambda)
        cnt = cnt + 1
        dcuts = c(dcuts, val)
    }
    Y = matrix(0, n, p)
    oldval = min(X)
    for (i in 1:length(dcuts)) {
        val = quantile(X, dcuts[i])
        Y[which(X < val & X >= oldval)] = i - 1
        oldval = val
    }
    Y[X == max(X)] = max(Y) + 1
    return(Y)
}
```

`glmGeneric` *Generic Function for local log-linear graphical model.*

Description

(Generic) Function to implement the local Log-Linear Graphical Model based on pair-wise markov network with the efficient and parallel algorithm.

Usage

```
glmGeneric(X, Y = NULL, link, lambda, parallel = FALSE, nCpus = 4, standardize = TRUE)
```

Arguments

`X`	a pxn data matrix
`Y`	a qxn data matrix or NULL, defualt to NULL. If it's a data matrix, the column number (n) should be the same as X. Y should be of different distribution family as in X.
`link`	link family to specify the distribution family of the response. If Y is NULL, link indicates the family of X. If Y is not NULL, link indicates the family of Y.
`lambda`	lambda vector used for regularization
`parallel`	logical value to indicate if the process should be run in parallel on multiple threads, default to FALSE.
`nCpus`	number of (maximum) cores to use for parallel execution, default to 4.
`standardize`	Logical flag for x variable standardization, prior to fitting the model sequence. The coefficients are always returned on the original scale. Default is standardize=TRUE. This parameter passed to glmnet for the parameter of the same name.

Value

`ghat`	If a specific lambda is given, ghat is a pxp matrix of coefficients. If the lambda for the whole regularization path is input, a 3D (pxpx length of regularization path) matrix is returned, where `ghat[,,i]` is the coefficient matrix of the p variables for the `i`-lambda

References

N. Meinshausen and P. Buhlmann, 2006, High-dimensional graphs and variable selection with the lasso, *The Annals of Statistics*, **vol. 34**, no. 3, pp. 1436–1462.

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, Y = NULL, link, lambda, parallel = F, nCpus = 4,
    standardize = TRUE)
{
    if (is.null(Y)) {
        Z <- X
        p <- nrow(Z)
        q <- 0
    }
    if (!is.null(Y)) {
        if (ncol(X) == ncol(Y)) {
            Z <- rbind(X, Y)
            p = nrow(X)
            q = nrow(Y)
        }
    }
    if (length(lambda) > 1) {
        ghat = array(0, dim = c(nrow(Z), nrow(Z), length(lambda)))
        wrapper1 <- function(i) {
            tryCatch({
                fit = glmnet(t(Z[-i, ]), Z[i, ], family = link,
                  lambda = lambda, standardize = standardize)
            }, error = function(e) {
                fit = glmnetEmpty(t(Z[-i, ]), lambda)
            })
            fit$beta = as.matrix(fit$beta)
            if (ncol(fit$beta) < length(lambda)) {
                tmp = matrix(0, nrow = nrow(fit$beta), ncol = length(lambda))
                tmp[, 1:ncol(fit$beta)] = fit$beta
                tmp[, ncol(fit$beta):length(lambda)] = fit$beta[,
                  ncol(fit$beta)]
                fit$beta = tmp
            }
            if (i == 1) {
                ghat[i, 2:nrow(Z), ] = fit$beta
            }
            else if (i == nrow(Z)) {
                ghat[i, 1:(nrow(Z) - 1), ] = fit$beta
            }
            else {
                ghat[i, 1:(i - 1), ] = fit$beta[1:(i - 1), ]
                ghat[i, (i + 1):nrow(Z), ] = fit$beta[i:nrow(fit$beta),
```

```
                ]
            }
            return(ghat[i, , ])
        }
        if (parallel) {
            if (q == 0) {
                ghat2 = mclapply(1:nrow(Z), wrapper1)
                for (i in 1:nrow(Z)) {
                  ghat[i, , ] = ghat2[[i]]
                }
            }
            if (q != 0) {
                ghat2 = mclapply((p + 1):nrow(Z), wrapper1)
                for (i in (p + 1):nrow(Z)) {
                  ghat[i, , ] = ghat2[[i - p]]
                }
            }
            return(ghat)
        }
        if (parallel == F) {
            if (q == 0) {
                ghat2 = lapply(1:nrow(Z), wrapper1)
                for (i in 1:nrow(Z)) {
                  ghat[i, , ] = ghat2[[i]]
                }
            }
            if (q != 0) {
                ghat2 = lapply((p + 1):nrow(Z), wrapper1)
                for (i in (p + 1):nrow(Z)) {
                  ghat[i, , ] = ghat2[[i - p]]
                }
            }
            return(ghat)
        }
    }
    if (length(lambda) == 1) {
        ghat = matrix(0, nrow = nrow(Z), ncol = nrow(Z))
        if (parallel) {
            library(snowfall)
            sfInit(parallel = TRUE, cpus = nCpus)
            sfExport("X", local = T)
            sfExport("ghat", local = T)
            sfLibrary(glmnet)
            wrapper2 <- function(i) {
                tryCatch({
                  fit = glmnet(t(Z[-i, ]), Z[i, ], family = link,
                    lambda = lambda, standardize = standardize)
                }, error = function(e) {
                  fit = glmnetEmpty(t(Z[-i, ]), lambda)
                })
                fit$beta = as.numeric(fit$beta)
                if (i == 1) {
                  ghat[i, 2:nrow(Z)] = fit$beta
```

```
            }
            else if (i == nrow(Z)) {
              ghat[i, 1:(nrow(Z) - 1)] = fit$beta
            }
            else {
              ghat[i, 1:(i - 1)] = fit$beta[1:(i - 1)]
              ghat[i, (i + 1):nrow(Z)] = c(fit$beta[i:length(fit$beta)])
            }
            return(ghat[i, ])
        }
        if (q == 0) {
            sfExport("wrapper2")
            ghat = sfSapply(1:nrow(Z), wrapper2)
            sfStop()
        }
        if (q != 0) {
            sfExport("wrapper2")
            ghat = sfSapply((p + 1):nrow(Z), wrapper2)
            sfStop()
        }
        return(ghat)
    }
    if (parallel == F) {
        st = p + 1
        if (q == 0) {
            st = 1
        }
        for (i in st:nrow(Z)) {
            tryCatch({
              fit = glmnet(t(Z[-i, ]), Z[i, ], family = link,
                lambda = lambda, standardize = standardize)
            }, error = function(e) {
              fit = glmnetEmpty(t(Z[-i, ]), lambda)
            })
            fit$beta = as.numeric(fit$beta)
            if (i == 1) {
              ghat[i, 2:nrow(Z)] = fit$beta
            }
            else if (i == nrow(Z)) {
              ghat[i, 1:(nrow(Z) - 1)] = fit$beta
            }
            else {
              ghat[i, 1:(i - 1)] = fit$beta[1:(i - 1)]
              ghat[i, (i + 1):nrow(Z)] = c(fit$beta[i:length(fit$beta)])
            }
        }
        return(ghat)
    }
  }
}
```

glmnetEmpty *Empty local log-linear graphical model.*

Description

This function will return an null (empty) local log-linear graphical model.

Usage

```
glmnetEmpty(X, lambda)
```

Arguments

X	a pxn data matrix
lambda	lambda vector used for regularization

Value

fit	a matrix of coefficients with zero values, in the dimension of (number of lambda) x p (number of variable)

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, lambda)
{
    fit = list()
    fit$a0 <- rep(0, length(lambda))
    fit$lambda <- lambda
    fit$df <- 0
    fit$dim <- c(ncol(X), length(lambda))
    fit$beta <- Matrix(0, ncol(X), length(lambda))
    rownames(fit$beta) <- colnames(X)
    colnames(fit$beta) <- paste("s", 0:(length(lambda) - 1),
        sep = "")
    return(fit)
  }
```

`glmpois` *Poisson based neighborhood selection*

Description

Poisson based neighborhood selection with X (pxn) on a fixed regularization path.

Usage

```
glmpois(X, lambda, parallel = FALSE, nCpus = 4)
```

Arguments

`X`	a pxn data matrix
`lambda`	regularization parameter, could be a single numerical value or a vector of numeric values (for the whole regularization path)
`parallel`	logical value to indicate if the process should be run parallelly in multiple threads, default to FALSE
`nCpus`	number of (maximum) cores to be used for parallel execution, default to 4

Details

This function will depends on the `glmnet` function. The neighborhood selection method is based on Meinshausen and Buhlmann neighborhood selection methods proposed for Gaussian graphical models.

Value

`ghat`	If a specific lambda is given, ghat is a pxp matrix of coefficients. If the lambda for the whole regularization path is input, a 3D (pxpx length of regularization path) matrix is returned, where `ghat[,,i]` is the coefficient matrix of the p variables for the `i`-lambda

References

N. Meinshausen and P. Buhlmann, 2006, High-dimensional graphs and variable selection with the lasso, *The Annals of Statistics*, **vol. 34**, no. 3, pp. 1436–1462.

See Also

`glmnet`

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, lambda, parallel = F, nCpus = 4)
{
    if (length(lambda) > 1) {
        ghat = array(0, dim = c(nrow(X), nrow(X), length(lambda)))
        if (parallel) {
            wrapper <- function(i) {
                fit = glmnet(t(X[-i, ]), X[i, ], family = "poisson",
                  lambda = lambda)
                fit$beta = as.matrix(fit$beta)
                if (ncol(fit$beta) < length(lambda)) {
                  tmp = matrix(0, nrow = nrow(fit$beta), ncol = length(lambda))
                  tmp[, 1:ncol(fit$beta)] = fit$beta
                  tmp[, ncol(fit$beta):length(lambda)] = fit$beta[,
                    ncol(fit$beta)]
                  fit$beta = tmp
                }
                if (i == 1) {
                  ghat[i, 2:nrow(X), ] = fit$beta
                }
                else if (i == nrow(X)) {
                  ghat[i, 1:(nrow(X) - 1), ] = fit$beta
                }
                else {
                  ghat[i, 1:(i - 1), ] = fit$beta[1:(i - 1),
                    ]
                  ghat[i, (i + 1):nrow(X), ] = fit$beta[i:nrow(fit$beta),
                    ]
                }
                return(ghat[i, , ])
            }
            library(multicore)
            ghat2 = mclapply(1:nrow(X), wrapper)
            for (i in 1:nrow(X)) {
                ghat[i, , ] = ghat2[[i]]
            }
            return(ghat)
        }
        if (parallel == F) {
            wrapper <- function(i) {
                fit = glmnet(t(X[-i, ]), X[i, ], family = "poisson",
                  lambda = lambda)
                fit$beta = as.matrix(fit$beta)
                if (ncol(fit$beta) < length(lambda)) {
                  tmp = matrix(0, nrow = nrow(fit$beta), ncol = length(lambda))
                  tmp[, 1:ncol(fit$beta)] = fit$beta
                  tmp[, ncol(fit$beta):length(lambda)] = fit$beta[,
```

```
                ncol(fit$beta)]
              fit$beta = tmp
            }
            if (i == 1) {
              ghat[i, 2:nrow(X), ] = fit$beta
            }
            else if (i == nrow(X)) {
              ghat[i, 1:(nrow(X) - 1), ] = fit$bet
            }
            else {
              ghat[i, 1:(i - 1), ] = fit$beta[1:(i - 1),
                ]
              ghat[i, (i + 1):nrow(X), ] = fit$beta[i:nrow(fit$beta),
                ]
            }
            return(ghat[i, , ])
        }
        ghat2 = lapply(1:nrow(X), wrapper)
        for (i in 1:nrow(X)) {
            ghat[i, , ] = ghat2[[i]]
        }
        return(ghat)
    }
}
if (length(lambda) == 1) {
    ghat = matrix(0, nrow = nrow(X), ncol = nrow(X))
    if (parallel) {
        library(snowfall)
        sfInit(cpus = nCpus)
        sfExport("X", local = T)
        sfExport("ghat", local = T)
        sfLibrary(glmnet)
        wrapper <- function(i) {
            fit = glmnet(t(X[-i, ]), X[i, ], family = "poisson",
              lambda = lambda)
            fit$beta = as.numeric(fit$beta)
            if (i == 1) {
              ghat[i, 2:nrow(X)] = fit$beta
            }
            else if (i == nrow(X)) {
              ghat[i, 1:(nrow(X) - 1)] = fit$beta
            }
            else {
              ghat[i, 1:(i - 1)] = fit$beta[1:(i - 1)]
              ghat[i, (i + 1):nrow(X)] = c(fit$beta[i:length(fit$beta)])
            }
            return(ghat[i, ])
        }
        sfExport("wrapper")
        ghat = sfSapply(1:nrow(X), wrapper)
        sfStop()
        return(ghat)
    }
```

```
        for (i in 1:nrow(X)) {
            fit = glmnet(t(X[-i, ]), X[i, ], family = "poisson",
                lambda = lambda)
            fit$beta = as.numeric(fit$beta)
            if (i == 1) {
                ghat[i, 2:nrow(X)] = fit$beta
            }
            else if (i == nrow(X)) {
                ghat[i, 1:(nrow(X) - 1)] = fit$beta
            }
            else {
                ghat[i, 1:(i - 1)] = fit$beta[1:(i - 1)]
                ghat[i, (i + 1):nrow(X)] = c(fit$beta[i:length(fit$beta)])
            }
        }
        return(ghat)
    }
}
```

GMS *Local Log-linear Graphical Models*

Description

This class of objects is returned by various "GM" functions included in this expMRF package, to represent the fitted markov networks over the regularization paths. Objects of this class have the print method to display the core information of the fitted models and plot method to plot the optimal markov network.

Arguments

v	vector of (nlams) variability measured from the stability selection
lambda.path	vector lambda used for regularization
opt.lambda	lambda value that gives the optimal network (network with maximum variability)
network	a list of pxp coefficient matrix along the regularization.
opt.index	index of the regularization value that gives the optimal network

See Also

LPGM.select, SPGM.select, TPGM.select, LGGM.select, LISM.select

`lambdaMax` *Maximum lambda*

Description

Compute the maximum lambda

Usage

```
lambdaMax(X)
```

Arguments

`X`	nxp data matrix

Details

Largest value for regularization (maximum lambda), which is the maximum element from X'X

Value

an integer value

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X)
{
    tmp = t(X) %*% X
    return(max(tmp[upper.tri(tmp)]))
  }
```

`LGGM.select` *Local log-linear graphical model based on pair-wise Gaussian markov network.*

Description

Fitting the local log-Linear graphical model based on pair-wise Gaussian markov network using an efficient, parallel algorithm over a path of regularization parameters (lambda). This algorithm employs neighborhood selection in inferring network structure. Stability selection is used in selecting the optimal network.

Usage

```
LGGM.select(X, method = "LGGM", N = 100, beta = 0.05, lmin = 0.01, nlams = 20, lambda.path = NULL, paralle
```

Arguments

X	a pxn data matrix
method	specification of the variation of local log-linear Gaussian-based graphical model (LGGM), default to "LGGM".
N	number of iteration for stability selection, default to 100
beta	threshold value on sparsity of the network to filter out dense network
lmin	minimum lambda value, default to 0.01
nlams	number of lambda for regularization
lambda.path	vector lambda used for regularization
parallel	logical value to indicate if the process should be run parallelly in multiple threads, default to TRUE
nCpus	number of (maximum) cores to use for parallel execution, default to 4

Details

This function is more of the interface to model the local log-linear Gaussian markov network. Refer to LGM.select.generic for details in the model fitting.

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

See Also

LGM.select.generic, GMS

Examples

```
library(expMRF)
n = 100
p = 50
tData.G = matrix(rnorm(n*p),n,p)
lggm.path.all.p = LGGM.select(t(tData.G), nlams=10, N=10, beta=0.05, nCpus=2, parallel=TRUE)
lggm.path.all.p
```

`LGM.select.generic` *Generic local log-linear graphical model*

Description

A generic function allow the fitting of the local log-Linear graphical model based on pair-wise markov properties using an efficient, parallel algorithm over a path of regularization parameters (lambda). This algorithm employs neighborhood selection in inferring network structure. Stability selection is used in selecting the optimal network.

Usage

```
LGM.select.generic(X, method = "LPGM", link = "poisson", N = 100, beta = 0.05, lmin = 0.01, nlams = 20, la
```

Arguments

`X`	a pxn data matrix
`method`	specification of the variation of local log-linear graphical model to be fitted. Default to pair-wise Poisson graphical models (LPGM). Other methods allowed included "TPGM" fro truncated Poisson, "SPGM" for sublinear Poisson, "LGGM" for Gaussian, "LISM" for Ising model (binomial).
`link`	specification of the exponential family of the data matrix X. Default to Poisson distribution ("poisson"). Other links allowed "gaussian" for Gaussian, "binomial" for binary data.
`N`	number of iteration for stability selection, default to 100
`beta`	threshold value on sparsity of the network to filter out dense network, default to 0.05
`lmin`	minimum lambda value, default to 0.01
`nlams`	number of lambda for regularization, default to 20
`lambda.path`	vector lambda used for regularization, defaults to NULL
`parallel`	logical value to indicate if the process should be run parallelly in multiple threads, default to TRUE
`nCpus`	number of (maximum) cores to use for parallel execution, default to 4

Details

This is the generic function to implement the local log-Linear markov network (proposed in the first reference below). The core function of this funciton is to employ neighborhood selection to infer the network structure (`glmGeneric`) and STAR (stability selection) for various distribution families.

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

See Also

GMS, glmGeneric, myglmnet.max

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, method = "LPGM", link = "poisson", N = 100, beta = 0.05,
    lmin = 0.01, nlams = 20, lambda.path = NULL, parallel = T,
    nCpus = 4)
{
    require("huge")
    require("glmnet")
    if (is.null(lambda.path)) {
        lmax = myglmnet.max(X, link = link)
        lambda.path = exp(seq(log(lmax), log(lmin), l = nlams))
    }
    if (parallel == T) {
        b = min(c(10 * sqrt(ncol(X)), 0.8 * ncol(X)))
        ghat = list()
        ghat.path = list()
        ghat.path$path = vector("list", length(lambda.path))
        v = c()
        for (i in 1:N) {
            cat(paste(method, ": Conducting sampling ... in progress: ",
                floor(100 * (i/N)), "%", collapse = ""), "\r")
            flush.console()
            glmpois.good <- 1
            while (glmpois.good) {
                good <- 1
                while (good) {
                  index = sample(1:ncol(X), b, replace = F)
                  if (sum(apply(X[, index], 1, function(x) length(unique(x)) ==
                    1)) == 0) {
                    good <- 0
                  }
                }
                tryCatch({
                  ghat.path$raw = glmGeneric(X[, index], NULL,
                    link = link, lambda = lambda.path, parallel = T,
```

```
                nCpus = nCpus)
              glmpois.good <- 0
            }, error = function(e) {
              cat("glmnet returns empty model. Try again.")
            })
        }
        for (j in 1:length(lambda.path)) {
            tmp = ghat.path$raw[, , j]
            tmp[abs(tmp) < 1e-06] = 0
            tmp[abs(tmp) > 1e-06] = 1
            diag(tmp) = 0
            if (is.null(ghat.path$path[[j]])) {
              ghat.path$path[[j]] = tmp
            }
            else {
              ghat.path$path[[j]] = ghat.path$path[[j]] +
                tmp
            }
        }
    }
    for (i in 1:length(lambda.path)) {
        D = ghat.path$path[[i]]
        D = D/N
        D = 2 * D * (1 - D)
        v = c(v, mean(D[upper.tri(D)]))
    }
    v = cummax(v)
    ghat$v = v
    ghat$lambda.path = lambda.path
    ghat$opt.lambda = lambda.path[which(v == max(v[v < beta]))]
    ghat$network = glmGeneric(X, NULL, link = link, lambda = lambda.path,
        parallel = T, nCpus = nCpus)
    ghat$network = lapply(1:nlams, function(r) {
        return(ghat$network[, , r])
    })
    ghat$opt.index = which(v == max(v[v < beta]))
    ghat$call <- match.call()
    cat(paste("\n", method, " Completed.", "\n", sep = ""))
    class(ghat) <- "GMS"
    return(ghat)
}
if (parallel == F) {
    b = min(c(10 * sqrt(ncol(X)), 0.8 * ncol(X)))
    ghat = list()
    v = c()
    for (j in 1:length(lambda.path)) {
        cat(paste(method, ": Conducting sampling ... in progress: ",
            floor(100 * (j/length(lambda.path))), "%", collapse = ""),
            "\r")
        flush.console()
        D = matrix(0, nrow = nrow(X), ncol = nrow(X))
        for (i in 1:N) {
            glmpois.good <- 1
```

```
                while (glmpois.good) {
                  good <- 1
                  while (good) {
                    index = sample(1:ncol(X), b, replace = F)
                    if (sum(apply(X[, index], 1, function(x) length(unique(x)) ==
                      1)) == 0) {
                      good <- 0
                    }
                  }
                  tryCatch({
                    tmp = glmGeneric(X[, index], NULL, link = link,
                      lambda = lambda.path[j], parallel = F)
                    glmpois.good <- 0
                  }, error = function(e) {
                    cat("glmnet returns empty model. Try again.\n")
                  })
                }
                tmp[abs(tmp) < 1e-06] = 0
                tmp[abs(tmp) > 1e-06] = 1
                D = D + tmp
            }
            D = D/N
            D = 2 * D * (1 - D)
            v = c(v, mean(D[upper.tri(D)]))
        }
        v = cummax(v)
        ghat$v = v
        ghat$lambda.path = lambda.path
        ghat$opt.lambda = lambda.path[which(v == max(v[v < beta]))]
        ghat$network = glmGeneric(X, NULL, link = link, lambda = lambda.path,
            parallel = parallel, nCpus = nCpus)
        ghat$network = lapply(1:nlams, function(r) {
            return(ghat$network[, , r])
        })
        ghat$opt.index = which(v == max(v[v < beta]))
        ghat$call <- match.call()
        cat(paste("\n", method, " Completed.", "\n", sep = ""))
        class(ghat) <- "GMS"
        return(ghat)
    }
}
```

LISM.select *Local log-linear graphical model based on Ising model.*

Description

Fitting the Ising model through the efficient and parallel local log-linear graphical model algorithm over a path of regularization parameters. Stability selection is used in selecting the optimal network.

Usage

```
LISM.select(X, method = "LISM", N = 100, beta = 0.05, lmin = 0.01, nlams = 20, lambda.path = NULL, paralle
```

Arguments

X	a pxn data matrix
method	specification of the variation of local Ising-model (LISM), default to "LISM".
N	number of iteration for stability selection, default to 100
beta	threshold value on sparsity of the network to filter out dense network
lmin	minimum lambda value, default to 0.01
nlams	number of lambda for regularization
lambda.path	vector lambda used for regularization
parallel	logical value to indicate if the process should be run parallelly in multiple threads, default to TRUE
nCpus	number of (maximum) cores to use for parallel execution, default to 4

Details

This function is more of the interface to model the Ising model. Refer to LGM.select.generic for details in the model fitting.

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

See Also

LGM.select.generic, GMS

Examples

```
library(expMRF)

n = 100
p = 50
tData.Y = matrix(rbinom(n*p,1,0.5),n,p)
lism.path.all.p = LISM.select(t(tData.Y), nlams=10, N=10, beta=0.05, nCpus=2, parallel=TRUE)
lism.path.all.p


##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, method = "LISM", N = 100, beta = 0.05, lmin = 0.01,
    nlams = 20, lambda.path = NULL, parallel = T, nCpus = 4)
```

```
{
    ghat <- LGM.select.generic(X, method = method, link = "binomial",
        N = N, beta = beta, lmin = lmin, nlams = nlams, lambda.path = lambda.path,
        parallel = parallel, nCpus = nCpus)
    if (!is.null(ghat)) {
        ghat$call <- match.call()
    }
    return(ghat)
  }
```

`LPGM.select` *Log-Linear Graphical Model based on Pair-wise Poisson Markov Network*

Description

Fitting the local Log-Linear Graphical Model based on pair-wise Poisson Markov Network using an efficient, parallel algorithm named Poisson Graphical Lasso over a path of regularization parameters (lambda). This algorithm employs neighborhood selection in inferring network structure. Stability selection is used in selecting the optimal network.

Usage

```
LPGM.select(X, method = "LPGM", N = 100, beta = 0.05, lmin = 0.01, nlams = 20, lambda.path = NULL, paralle
```

Arguments

`X`	a pxn data matrix
`method`	specification of the variation of log-linear Poisson-based graphical model (LPGM), default to "LPGM". Other two methods allowed are truncated poisson graphical model (TPGM) and sub-linear poisson graphical model (SPGM).
`N`	number of iteration for stability selection, default to 100
`beta`	threshold value on sparsity of the network to filter out dense network
`lmin`	minimum lambda value, default to 0.01
`nlams`	number of lambda for regularization
`lambda.path`	vector lambda used for regularization
`parallel`	logical value to indicate if the process should be run parallelly in multiple threads, default to TRUE
`nCpus`	number of (maximum) cores to use for parallel execution, default to 4

Details

This function is more of the interface to model the local log-linear Poisson markov network. Refer to `LGM.select.generic` for details in the model fitting.

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

See Also

LGM.select.generic, GMS

Examples

```
library(huge)
library(expMRF)
n = 200
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace=TRUE), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

#-# Run LPGM
lpgm.path.all = LPGM.select(t(simData), nlams=20, N=10, beta=0.05, parallel=FALSE)
str(lpgm.path.all)

##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, method = "LPGM", N = 100, beta = 0.05, lmin = 0.01,
    nlams = 20, lambda.path = NULL, parallel = T, nCpus = 4)
{
    ghat <- LGM.select.generic(X, method = method, link = "poisson",
        N = N, beta = beta, lmin = lmin, nlams = nlams, lambda.path = lambda.path,
        parallel = parallel, nCpus = nCpus)
    if (!is.null(ghat)) {
        ghat$call <- match.call()
    }
    return(ghat)
  }
```

`Merge.GraphXY` *Merge Graphs*

Description

Internal function to merge the two inferred markov networks produced by mixed-grpahical model (MGM.select function)

Usage

```
Merge.GraphXY(network.X, network.Y, namesX, namesY, nlams, method = "Both")
```

Arguments

`network.X`	the first network object inferred
`network.Y`	the second network object inferred
`namesX`	the (p) variable names from data matrix X
`namesY`	the (q) variable naems from data matrix Y
`nlams`	number of lambda for regularization
`method`	specification of the variation of methods in inferring the network, default to "Both", two other methods allowed are "Right" and "Left". Refer to `MGM.select` for details.

Value

A list of nlams merged networks with dimension (p+q) x (p+q); given X has p variables and Y has q variables.

See Also

`MGM.select`

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (network.X, network.Y, namesX, namesY, nlams, method = "Both")
{
    if (method == "Both") {
        results <- lapply(1:nlams, function(r) {
            tmp1 <- network.X[, , r]
            rownames(tmp1) <- colnames(tmp1) <- c(namesX, namesY)
            tmp2 <- network.Y[, , r]
            rownames(tmp2) <- colnames(tmp2) <- c(namesY, namesX)
```

```
            tmp = tmp1
            tmp[namesX, colnames(tmp)] = tmp2[namesX, colnames(tmp)]
            return(tmp)
        })
    }
    if (method == "Right") {
        results <- lapply(1:nlams, function(r) {
            tmp1 <- network.X[, , r]
            rownames(tmp1) <- colnames(tmp1) <- c(namesX, namesY)
            tmp2 <- network.Y[, , r]
            rownames(tmp2) <- colnames(tmp2) <- namesX
            tmp = tmp1
            tmp[namesX, namesX] = tmp2[namesX, namesX]
            return(tmp)
        })
    }
    if (method == "Left") {
        results <- lapply(1:nlams, function(r) {
            tmp1 <- network.X[, , r]
            rownames(tmp1) <- colnames(tmp1) <- namesY
            tmp2 <- network.Y[, , r]
            rownames(tmp2) <- colnames(tmp2) <- c(namesY, namesX)
            tmp = tmp2
            tmp[namesY, namesY] = tmp1[namesY, namesY]
            return(tmp)
        })
    }
    return(results)
}
```

`MGM.select` *Mixed-Graphical Model*

Description

Fitting the local Log-linear Graphical model based on pair-wise markov properties from a data matrix with combination of two data types (poisson and binary).

Usage

```
MGM.select(X, Y, xlink = "poisson", ylink = "binomial", method = "Both", N = 100, beta = 0.05,
lmin = 0.01, nlams = 20, lambda.path = NULL, parallel = TRUE, nCpus = 4, standardize = TRUE)
```

Arguments

\ itemXa pxn data matrix. \ itemYa qxn data matrix. \ itemxlinkthe distribution family for data X, default to "poisson" \ itemylinkthe distribution family for data Y, default to "binomial" \ itemmethodspecification of the variation of methods in inferring the network between two data types. Refer details section below for more discussion. \ itemNnumber of iteration for stability selection, default to

100 \ itembetathreshold value on sparsity of the network to filter out dense network \ itemlminminimum lambda value, default to 0.01 \ itemnlamsnumber of lambda for regularization \ itemlambda.pathvector lambda used for regularization \ itemparallellogical value to indicate if the process should be run parallelly in multiple threads, default to TRUE \ itemnCpusnumber of (maximum) cores to use for parallel execution, default to 4 \ itemstandardizeLogical flag for x variable standardization, prior to fitting the model sequence. The coefficients are always returned on the original scale. Default is standardize=TRUE. This parameter passed to glmnet for the parameter of the same name.

Details

To infer network of the data matrix Z (combined data from X and Y by row), three alternative methods are implemented: method = "Both", mixed MRF of Z (or X <-> Y); which are performing two CRF regressions: `X_i ~ X_{\ i} + Y` and `Y_i ~ X + Y_{\ i}` method = "Right"; which are which are performing a MRF regression on X: `X_i ~ X_{\ i}` and CRF on Y `Y_i ~ X + Y_{\ i}` method = "Left"; which are which are performing a CRF regression on X: `X_i ~ X_{\ i} + Y` and MRF on Y `Y_i ~ Y_{\ i}` For each method, MRF or CRF are implemented separately and then only merge to form one final inferred network.

Also, the sample size of both data X and Y should be the same and they should be arranged into the same order.

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

See Also

`Merge.GraphXY`

Examples

```
n = 100
p = 100
q = 50

tData.X = matrix(rpois(n*p,1),n,p)
tData.Y = matrix(rbinom(n*q,1,0.5),n,q)

fixpath <- exp(seq(log(10),log(0.01),l=10))

mgm.nets <- MGM.select(t(tData.X),t(tData.Y),xlink="poisson", ylink="binomial", method="Both", N=10,beta=0.05, l
mgm.nets.right <- MGM.select(t(tData.X),t(tData.Y),xlink="poisson", ylink="binomial", method="Right", N=10,beta=
mgm.nets.left <- MGM.select(t(tData.X),t(tData.Y),xlink="poisson", ylink="binomial", method="Left", N=10,beta=0.
```

myglmnet.max *Maximum lambda from binary search*

Description

Obtain the regularization paramter lambda through binary search between zero to the maximum of X'X, in search for the smallest value that gives a null graphical model (empty network).

Usage

```
myglmnet.max(X, link = "poisson", delta = 0.01)
```

Arguments

X	a pxn data matrix
link	specification of the exponential family of the data matrix X. Default to Poisson distribution ("poisson"). Other links allowed "gaussian" for Gaussian, "binomial" for binary data.
delta	shift-size for the binary search, default to 0.01

Value

numeric value for regularization parameter that will return a null model: the maximum lambda.

See Also

lambdaMax, glmGeneric

Examples

```
library(PGM)
library(huge)
library(glmnet)
n = 200
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace =TRUE), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

lmax = myglmnet.max(t(simData))


##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, link = "poisson", delta = 0.01)
```

```
{
    minlambda = 0
    maxlambda = lambdaMax(t(X))
    while (1) {
        mid = (minlambda + maxlambda)/2
        tmp = glmGeneric(X, NULL, link = link, lambda = mid)
        tmp[abs(tmp) < 1e-06] = 0
        tmp[abs(tmp) > 1e-06] = 1
        if (sum(tmp) > 0) {
            minlambda = mid + delta
        }
        else {
            maxlambda = mid - delta
        }
        if (abs(maxlambda - minlambda) < delta) {
            return(mid)
        }
    }
  }
```

`plot.GMS` *Plot GMS object.*

Description

Default function to plot the optimal network of the GMS object (optimal markov network over the regularization path)

Usage

```
plot.GMS(x, fn = "", th = 1e-06, ...)
```

Arguments

`x`	a GMS object
`fn`	file name to save the network plot. Default to be an empty string, so the network is plotted to the standard output (screen). NOTE: if a filename is specificed, it should be filename for PDF file.
`th`	numeric value, default to 1e-06. To specify the threshold if the estimated coefficient between two variables is to be considered connected.

Details

This is the default plotting function for GMS objects (markov networks inferred over a regularization path). Refer to GMS for details on GMS object. The function will plot the optimal network on the screen by default. However, given a filename, the plot will be saved to a PDF file. The optimal netowrk will be plotted in force-directed layout (layout.kamada.kawai with default parameters implemented in igraph package).

See Also

GMS

Examples

```
n = 100
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace =T), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

# Run LPGM
lpgm.path.all.p = LPGM.select(t(simData), nlams=10, N=10, beta=0.05, nCpus=2, parallel=T)
lpgm.path.all.p

plot(lpgm.path.all.p, fn="lpgm.opt.net.pdf")
```

---

print.GMS *Print the GMS object.*

Description

Default function to print the GMS object to the standard output.

Usage

```
print.GMS(x, ...)
```

Arguments

x	a GMS object

Details

This is the default print function to display information of a GMS object to the standard output in readable format. It will print the information of the function called to create the object, the index of the optimal network, the whole regularization path, estimated variabilities over the regularization path, and list of inferred markov networks over the whole regularization path. Refer to GMS for details on GMS object.

See Also

GMS

Examples

```
n = 100
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace =T), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

# Run LPGM
lpgm.path.all.p = LPGM.select(t(simData), nlams=10, N=10, beta=0.05, nCpus=2, parallel=T)

# Call this default print function
lpgm.path.all.p
```

SPGM.select *Log-Linear Graphical Model based on Pair-wise Sub-linear truncated Poisson Markov Network*

Description

Fitting the local Log-Linear Graphical Model based on pair-wise sublinear-truncated Poisson Markov Network. The network modeling algorithm is the same as LPGM.

Usage

```
SPGM.select(X, R, R0 = 0, N = 100, beta = 0.05, lmin = 0.01, nlams = 20, lambda.path = NULL, parallel = TRU
```

Arguments

X	a pxn data matrix
R	lower-bound threshold value for the trunctation, has to be positive
R0	lower-bound threshold value for the trunctation, default to 0
N	number of iteration for stability selection, default to 100
beta	threshold value on sparsity of the network to filter out dense network
lmin	minimum lambda value, default to 0.01
nlams	number of lambda for regularization
lambda.path	vector lambda used for regularization
parallel	logical value to indicate if the process should be run parallelly in multiple threads, default to TRUE
nCpus	number of (maximum) cores to use for parallel execution, default to 4

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

See Also

GMS, Bsublin, LPGM.select

Examples

```
library(PGM)
library(huge)
n = 200
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace=TRUE), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

range(simData)
spgm.path.all.p = SPGM.select(t(simData), 4, 2, nlams=20, N=10, beta=0.05, nCpus=2, parallel=TRUE)
spgm.path.all.p


##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, R, R0 = 0, N = 100, beta = 0.05, lmin = 0.01, nlams = 20,
    lambda.path = NULL, parallel = T, nCpus = 4)
{
    require("huge")
    require("glmnet")
    if (R < 0) {
        cat("ERROR: Truncating threshold R should be positive. \n")
        ghat = NULL
        return(ghat)
    }
    Xorig <- X
    X <- round(Bsublin(X, R, R0))
    ghat <- LPGM.select(X, method = "SPGM", N = N, beta = beta,
        lmin = lmin, nlams = nlams, lambda.path = lambda.path,
        parallel = parallel, nCpus = nCpus)
    if (!is.null(ghat)) {
        ghat$call = match.call()
    }
    return(ghat)
  }
```

TPGM.select *Log-Linear Graphical Model based on Pair-wise truncated Poisson Markov Network*

Description

Fitting the local Log-Linear Graphical Model based on pair-wise truncated Poisson Markov Network. The network modeling algorithm is the same as LPGM.

Usage

```
TPGM.select(X, R, N = 100, beta = 0.05, lmin = 0.01, nlams = 20, lambda.path = NULL, parallel = TRUE, nCpu
```

Arguments

X	a pxn data matrix
R	threshold value for the trunctation, has to be positive
N	number of iteration for stability selection, default to 100
beta	threshold value on sparsity of the network to filter out dense network
lmin	minimum lambda value, default to 0.01
nlams	number of lambda for regularization
lambda.path	vector lambda used for regularization
parallel	logical value to indicate if the process should be run parallelly in multiple threads, default to TRUE
nCpus	number of (maximum) cores to use for parallel execution, default to 4

Details

In truncation, the elements in data matrix X will be set to R if the value is greater than R.

Value

an object of class GMS object will be returned, represents the modeled markov networks over the regularization path. See GMS for details.

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

See Also

GMS, LPGM.select

Examples

```
library(PGM)
library(huge)
n = 200
p = 50
gdata = huge.generator(n,d=p, graph="scale-free",v=0.1,u=0.01)
smatrix  = matrix(sample(c(1,-1), nrow(gdata$theta)*ncol(gdata$theta), replace=TRUE), nrow = nrow(gdata$theta) )
simData = WPGMSim(n,p,R=10, alpha = rep(0,p), Theta = 0.1*as.matrix(gdata$theta)*smatrix, maxit = 100 )

range(simData)
tpgm.path.all.p5 = TPGM.select(t(simData), 5, nlams=20, N=10, beta=0.05, nCpus=2, parallel=TRUE)
tpgm.path.all.p5


##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, R, N = 100, beta = 0.05, lmin = 0.01, nlams = 20,
    lambda.path = NULL, parallel = T, nCpus = 4)
{
    require("huge")
    require("glmnet")
    if (R < 0) {
        cat("ERROR: Truncating threshold R should be positive. \n")
        ghat = NULL
        return(ghat)
    }
    Xorig <- X
    X[X > R] <- R
    ghat <- LPGM.select(X, method = "TPGM", N = N, beta = beta,
        lmin = lmin, nlams = nlams, lambda.path = lambda.path,
        parallel = parallel, nCpus = nCpus)
    if (!is.null(ghat)) {
        ghat$call = match.call()
    }
    return(ghat)
  }
```

`WPGM.neighborhood` *WPGM neighborhood*

Description

WPGM neighborhood selection problem (on one lambda)

Usage

```
WPGM.neighborhood(X, Y, R, lam, startb = 0)
```

Arguments

X	a nxp data matrix
Y	nx1 vector of responses (Poisson?)
R	threshold value for truncating
lam	numeric lambda value (regularization parameter)
startb	default to 0, otherwise a starting vector for beta

Value

A list of:

alpha	intercept
beta	vector of p coefficients

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, Y, R, lam, startb = 0)
{
    n = nrow(X)
    p = ncol(X)
    thr = 1e-08
    maxit = 1e+06
    Xt = cbind(t(t(rep(1, n))), X)
    if (sum(startb) == 0) {
        bhat = matrix(rnorm(p + 1) * 0.01, p + 1, 1)
    }
    else {
        bhat = startb
    }
    step = 0.1
    ind = 1
    iter = 1
    while (thr < ind & iter < maxit) {
        oldb = bhat
        t = 1
        grad = wpgmGrad(Xt, Y, R, oldb)
        oldobj = wpgmObj(Xt, Y, R, oldb)
        tmp = oldb - t * grad
        bhat[1] = tmp[1]
        bhat[-1] = sign(tmp[-1]) * sapply(abs(tmp[-1]) - lam *
            t, max, 0)
        while (wpgmObj(Xt, Y, R, bhat) > oldobj - t(grad) %*%
            (oldb - bhat) + sum((oldb - bhat)^2)/(2 * t)) {
            t = t * step
            tmp = oldb - t * grad
```

```
            bhat[1] = tmp[1]
            bhat[-1] = sign(tmp[-1]) * sapply(abs(tmp[-1]) -
                lam * t, max, 0)
        }
        iter = iter + 1
        ind = sum((oldb - bhat)^2)/sum(oldb^2)
    }
    return(list(alpha = bhat[1], beta = bhat[-1]))
}
```

WPGM.network *Poisson network*

Description

Function to compute the poisson network over X

Usage

```
WPGM.network(X, R, nlams, lmin = 0.001, lambda = NULL, parallel = TRUE, ncores = 4)
```

Arguments

X	a pxn data matrix (of Poisson)
R	threshold value for truncating
nlams	number of lambdas for regularization path
lmin	minimum lambda value, default to 0.001
lambda	a vector of nlams lambda for whole regularization path, default to NULL
parallel	logical value to indicate if the network build should be run parallelly in multiple threads, default to TRUE
ncores	number of cores to use for parallel execution, default to 4

Value

A list of length of the regularization path, each element of the list represent the networks estimated over the regularization path. Each network is encoded in pxp matrix of coefficients.

See Also

WPGM.path.neighborhood

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, R, nlams, lmin = 0.001, lambda = NULL, parallel = T,
    ncores = 4)
{
    if (is.null(lambda)) {
        lmax = lambdaMax(t(X))
        lambda = exp(seq(log(lmax), log(lmin), l = nlams))
    }
    if (nlams != length(lambda)) {
        print("nlams is not equal to lams")
    }
    ghat = c()
    if (nlams > 0) {
        ghat = array(0, dim = c(nrow(X), nrow(X), length(lambda)))
    }
    wrapper <- function(i) {
        fit = WPGM.path.neighborhood(t(X[-i, ]), X[i, ], R, nlams,
            lambda = lambda, 0)
        fit$beta = as.matrix(fit$Bmat)
        if (i == 1) {
            ghat[i, 2:nrow(X), ] = fit$beta
        }
        else if (i == nrow(X)) {
            ghat[i, 1:(nrow(X) - 1), ] = fit$beta
        }
        else {
            ghat[i, 1:(i - 1), ] = fit$beta[1:(i - 1), ]
            ghat[i, (i + 1):nrow(X), ] = fit$beta[i:nrow(fit$beta),
                ]
        }
        return(ghat[i, , ])
    }
    ghat2 = c()
    if (parallel) {
        library(multicore)
        ghat2 = mclapply(1:nrow(X), wrapper, mc.cores = ncores)
    }
    else {
        ghat2 = lapply(1:nrow(X), wrapper)
    }
    for (i in 1:nrow(X)) {
        ghat[i, , ] = ghat2[[i]]
    }
    ghat = lapply(1:nlams, function(r) {
        return(ghat[, , r])
    })
    return(ghat)
```

```
}
```

`WPGM.path.neighborhood` *WPGM neighborhood over a regularization path*

Description

WPGM neighborhood selection problem over a grid of lambdas

Usage

```
WPGM.path.neighborhood(X, Y, R, nlams, lmin = 0.01, lambda = NULL, startb = 0)
```

Arguments

`X`	a nxp data matrix
`Y`	nx1 vector of responses (Poisson?)
`R`	threshold value for truncating
`nlams`	number of lambdas for regularization path (set nlams=1 to return form one value)
`lmin`	minimum lambda value, default to 0.01
`lambda`	a vector of nlams lambda, default to NULL
`startb`	default to 0, otherwise a starting vector for beta

Value

A list of:

`alphas`	1 x nlams vector of intercepts
`Bmat`	p x nlams sparse matrix of coefficients
`lambda`	the lambda values for regularization path

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, Y, R, nlams, lmin = 0.01, lambda = NULL, startb = 0)
{
    n = nrow(X)
    p = ncol(X)
    if (is.null(lambda)) {
        lmax = lambdaMax(t(X))
```

```
    lambda = exp(seq(log(lmax), log(lmin), l = nlams))
}
if (nlams == 1 & is.null(lambda)) {
    lambda = lmax
}
thr = 1e-08
maxit = 1e+06
Xt = cbind(t(t(rep(1, n))), X)
if (sum(startb) == 0) {
    bhat = matrix(rnorm(p + 1)/p, p + 1, 1)
}
else {
    bhat = startb
}
alphas = 0
Bmat = matrix(0, p, nlams)
step = 0.1
for (i in 1:nlams) {
    ind = 1
    iter = 1
    while (thr < ind & iter < maxit) {
        oldb = bhat
        t = 1
        grad = wpgmGrad(Xt, Y, R, oldb)
        oldobj = wpgmObj(Xt, Y, R, oldb)
        tmp = oldb - t * grad
        bhat[1] = tmp[1]
        bhat[-1] = sign(tmp[-1]) * sapply(abs(tmp[-1]) -
            lambda[i] * t, max, 0)
        newobj = wpgmObj(Xt, Y, R, bhat)
        while (newobj > 9999999 | is.na(newobj) | is.na(newobj)) {
            t = t/p
            tmp = oldb - t * grad
            bhat[1] = tmp[1]
            bhat[-1] = sign(tmp[-1]) * sapply(abs(tmp[-1]) -
              lambda[i] * t, max, 0)
            newobj = wpgmObj(Xt, Y, R, bhat)
        }
        while (newobj > oldobj - t(grad) %*% (oldb - bhat) +
            sum((oldb - bhat)^2)/(2 * t)) {
            t = t * step
            tmp = oldb - t * grad
            bhat[1] = tmp[1]
            bhat[-1] = sign(tmp[-1]) * sapply(abs(tmp[-1]) -
              lambda[i] * t, max, 0)
            newobj = wpgmObj(Xt, Y, R, bhat)
        }
        iter = iter + 1
        ind = sum((oldb - bhat)^2)
    }
    alphas[i] = bhat[1]
    Bmat[, i] = bhat[-1]
}
```

```
  return(list(alpha = alphas, Bmat = Bmat, lambda = lambda))
}
```

`WPGM.select` *Winsorized Poisson Graphical Model (WPGM)*

Description

Fitting the WPGM using efficient, parallel algorithm named Poisson Graphical Lasso. This algorithm employs neighborhood selection to infer network structure. Stability selection method "star" was used in selecting the optimal network.

Usage

```
WPGM.select(X, R=max(X), N=100, beta=0.05, lmin=0.0001, nlams=20,
lambda.path=NULL, parallel=FALSE, ncores = 4)
```

Arguments

`X`	pxn data matrix
`R`	threshold value for truncating, default to be the maximum of value of the input data matrix
`N`	number of iteration for stability selection, default to 100
`beta`	threshold value on sparsity of the network to filter out dense network
`lmin`	minimum lambda value, default to 0.0001
`nlams`	number of lambda for regularization
`lambda.path`	vector lambda used for regularization
`parallel`	logical value to indicate if the process should be run parallelly in multiple threads, default to FALSE
`ncores`	number of (maximum) cores to use for parallel execution, default to 4

Value

A list of five elements:

`v`	vector of (nlams) variability measured from the stability selection
`lambda.path`	vector lambda used for regularization
`opt.lambda`	lambda value that gives the optimal network (network with maximum variability)
`network`	a list of pxp coefficient matrix along the regularization.
`opt.index`	index of the regularization value that gives the optimal network

References

G.I. Allen and Z. Liu, 2012, A Log-Linear Graphical Model for Inferring Genetic Networks from High-Throughput Sequencing Data, *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2012)*.

E. Yang, P.K. Ravikumar, G.I. Allen, and Z. Liu, 2012, Graphical Models via Generalized Linear Models, *NIPS*, **vol. 25**, pp. 1367–1375.

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, R = max(X), method = "star", N = 100, beta = 0.05,
    lambda.path = NULL, nlams = 20, ncores = 4, parallel = F)
{
    if (is.null(lambda.path)) {
        lmax = lambdaMax(t(X))
        lambda.path = exp(seq(log(lmax), log(1e-04), l = nlams))
    }
    b = min(c(10 * sqrt(ncol(X)), 0.8 * ncol(X)))
    ghat = list()
    ghat.path = list()
    ghat.path$path = vector("list", length(lambda.path))
    v = c()
    for (i in 1:N) {
        cat(paste("WPGM: Conducting sampling ... in progress: ",
            floor(100 * (i/N)), "%", collapse = ""), "\r")
        flush.console()
        index = sample(1:ncol(X), b, replace = F)
        ghat.path$raw = WPGM.network(X[, index], R, nlams = length(lambda.path),
            lambda = lambda.path, parallel = parallel, ncores = ncores)
        for (j in 1:length(lambda.path)) {
            tmp = ghat.path$raw[[j]]
            tmp[abs(tmp) < 1e-06] = 0
            tmp[abs(tmp) > 1e-06] = 1
            diag(tmp) = 0
            if (is.null(ghat.path$path[[j]])) {
                ghat.path$path[[j]] = tmp
            }
            else {
                ghat.path$path[[j]] = ghat.path$path[[j]] + tmp
            }
        }
    }
    for (i in 1:length(lambda.path)) {
        D = ghat.path$path[[i]]
        D = D/N
        D = 2 * D * (1 - D)
        v = c(v, mean(D[upper.tri(D)]))
```

```
  }
  v = cummax(v)
  ghat$v = v
  ghat$lambda.path = lambda.path
  ghat$opt.lambda = lambda.path[which(v == max(v[v < beta]))]
  ghat$network = WPGM.network(X, R, nlams = length(lambda.path),
      lambda = lambda.path, parallel = T)
  ghat$opt.index = which(v == max(v[v < beta]))
  cat("\nWPGM Completed. \n")
  return(ghat)
}
```

WPGMSim *Winsorized PGM Gibbs Simulator*

Description

Winsorized PGM Gibbs Sampler (both positive and negatvie relationships)

Usage

```
WPGMSim(n, p, R, alpha, Theta, maxit = 10000)
```

Arguments

n	sample size
p	variable size
R	threshold value for truncating
alpha	a px1 vector
Theta	a pxp symmetric matrix (only off diags matter).
maxit	iterations for Gibbs sampler, default to 10000

Value

X	a nxp data matrix

Examples

```
wpgm.sim <- WPGMSim(10, 3, 2, rep(0.5, 3), matrix(-1, 3,3))
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (n, p, R, alpha, Theta, maxit = 10000)
{
    X = matrix(rpois(n * p, 1), n, p)
    iter = 1
    while (iter < maxit) {
```

```
            for (j in 1:p) {
                num = exp(matrix(1, n, 1) %*% t(alpha[j] * c(0:R) -
                    log(factorial(c(0:R)))) + matrix(c(0:R) %x% X[,
                    -j] %*% Theta[-j, j], n, R + 1))
                Pmat = num/matrix(apply(num, 1, sum), n, R + 1)
                X[, j] = apply(apply(Pmat, 1, mymult) == 1, 2, which) -
                    1
            }
            iter = iter + 1
        }
        return(X)
    }
```

Index