Package 'PGM'

March 5, 2014

Type Package

Title PGM: Poisson Graphical Models

Version 0.1

Date 2014-03-04

Author Ying-Wooi Wan, Zhandong Liu

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

Description This package contains methods to employ Poisson Graphical Models

License GPL2.0

LazyLoad yes

R topics documented:

PGM-package *Poisson Graphical Models*

Description

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

Details

Package:	PGM
Type:	Package
Version:	Trial.2.0
Date:	2014-03-04
License:	GPL2.0

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

Author(s)

Who wrote it

Maintainer: Who to complain to <yourfault@somewhere.net> ~~ 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> ~~

`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)
  }
```

glmpois	*Poisson based neighborhood selection*

Description

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

Usage

```
glmpois(X, lambda, parallel = F, 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)
        }
      }
```

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)]))
  }
```

`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, parallel = T, nCpus = 4)
```

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

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.

See Also

`myglmnet.max`, `glmpois`

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 )

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

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

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 Poisson graphical model (empty network).

Usage

```
myglmnet.max(X, delta = 0.01)
```

Arguments

`X`	a pxn data matrix
`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`, `glmpois`

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))
```

`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 = T, nCpus = 4)
```

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

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.

See Also

`Bsublin`, `LPGM.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 (X, R, R0 = 0, N = 100, beta = 0.05, lmin = 0.01, nlams = 20,
    lambda.path = NULL, parallel = T, nCpus = 4)
```

```
{
    if (R < 0) {
        cat("ERROR: Truncating threshold R should be positive. \n")
        ghat = NULL
        return(ghat)
    }
    Xorig <- X
    X <- round(Bsublin(X, R, R0))
    return(LPGM.select(X, method = "SPGM", N = N, beta = beta,
        lmin = lmin, nlams = nlams, lambda.path = lambda.path,
        parallel = parallel, nCpus = nCpus))
  }
```

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 = T, nCpus = 4)
```

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

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.

See Also

`LPGM.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 (X, R, N = 100, beta = 0.05, lmin = 0.01, nlams = 20,
    lambda.path = NULL, parallel = T, nCpus = 4)
{
    if (R < 0) {
        cat("ERROR: Truncating threshold R should be positive. \n")
        ghat = NULL
        return(ghat)
    }
    Xorig <- X
    X[X > R] <- R
    return(LPGM.select(X, method = "TPGM", N = N, beta = beta,
        lmin = lmin, nlams = nlams, lambda.path = lambda.path,
        parallel = parallel, nCpus = nCpus))
  }
```

`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 = T, 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=F, 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