**Contents**

R code for SparCC, CCLasso and Cosine similarity

SparCC

CCLasso

Cosine similarity

**SparCC:**

###############################################################################

# File: SparCC.R

# Aim : SparCC

#---------------------------------------------------------------------------------------------------------------------

# Author1 : Fang Huaying (Peking University)

# Email : hyfang@pku.edu.cn

# Date : 11/12/2014

# Author2 : Tianlu Chen (Shanghai Jiao Tong University)

# Email : chentianlu@sjtu.edu.cn

# Date : 02/22/2017

#---------------------------------------------------------------------------------------------------------------------

require(gtools);

#---------------------------------------------------------------------------------------------------------------------

# calculate both correlation r and pseudo p-values for SparCC by permutation

# function: SparCC.both

# input:

# x ------ nxp count data matrix, row is sample, col is variable

# n\_boot ------ Bootstrap times, Default: 20

# output: a list structure

# cor ------ correlation estimation

# p ------ pseudo p-values, Default: 0.5

#

# call functions SparCC.count and SparCC.frac

SparCC.both <- function (x,n\_boot = 20) {

c <- ncol(x);

n <- nrow(x);

cor = p = matrix(0,c, c);

result <- SparCC.count(x);

cor <- result$cor.w;

time\_name<-paste("time",1:n\_boot,sep="");

for (i in 1:n\_boot) {

# boot\_x <- matrix(sample(x,replace = T),n,c,byrow = T);

boot\_x <- sample(x,replace = T);

boot\_result <- SparCC.count(boot\_x);

boot\_cor <- boot\_result$cor.w;

# assign(time\_name[i],ifelse(abs(cor) <= abs(boot\_cor),1,0));

temp <-ifelse(abs(cor) <= abs(boot\_cor),1,0);

p<-p+temp

}

p <- p /n\_boot;

diag(p) <- 0;

return(list(cor = cor, p = p));

}

#---------------------------------------------------------------------------------------------------------------------

# SparCC for counts known

# function: SparCC.count

# input:

# x ------ nxp count data matrix, row is sample, col is variable

# imax ------ resampling times from posterior distribution. default 20

# kmax ------ max iteration steps for SparCC. default is 10

# alpha ------ the threshold for strong correlation. default is 0.1

# Vmin ------ minimal variance if negative variance appears. default is 1e-4

# output: a list structure

# cov.w ------ covariance estimation

# cor.w ------ correlation estimation

SparCC.count <- function(x, imax = 20, kmax = 10, alpha = 0.1, Vmin = 1e-4) {

# dimension for w (latent variables)

p <- as.numeric(ncol(x));

n <- as.numeric(nrow(x));

# posterior distribution (alpha)

x <- x + 1;

# store generate data

y <- matrix(0,n,p);

# store covariance/correlation matrix

cov.w <- cor.w <- matrix(0, p, p);

indLow <- lower.tri(cov.w, diag = T);

# store covariance/correlation for several posterior samples

covs <- cors <- matrix(0, p \* (p + 1) / 2, imax);

for(i in 1:imax) {

# generate fractions from posterior distribution

y <- t(apply(x, 1, function(x)

gtools::rdirichlet(n = 1, alpha = x)));

# estimate covariance/correlation

cov\_cor <- SparCC.frac(x = y, kmax = kmax, alpha = alpha, Vmin = Vmin);

# store variance/correlation only low triangle

covs[, i] <- cov\_cor$cov.w[indLow];

cors[, i] <- cov\_cor$cor.w[indLow];

}

# calculate median for several posterior samples

cov.w[indLow] <- apply(covs, 1, median);

cor.w[indLow] <- apply(cors, 1, median);

#

cov.w <- cov.w + t(cov.w);

diag(cov.w) <- diag(cov.w) / 2;

cor.w <- cor.w + t(cor.w);

diag(cor.w) <- 1;

#

return(list(cov.w = cov.w, cor.w = cor.w));

}

#---------------------------------------------------------------------------------------------------------------------

# SparCC for fractions known

# function: SparCC.frac

# input:

# x ------ nxp fraction data matrix, row is sample, col is variable

# kmax ------ max iteration steps for SparCC. default is 10

# alpha ------ the threshold for strong correlation. default is 0.1

# Vmin ------ minimal variance if negative variance appears. default is 1e-4

# output: a list structure

# cov.w ------ covariance estimation

# cor.w ------ correlation estimation

SparCC.frac <- function(x, kmax = 10, alpha = 0.1, Vmin = 1e-4) {

# Log transformation

x <- log(x);

p <- ncol(x);

# T0 = var(log(xi/xj)) variation matrix

TT <- stats::var(x);

T0 <- diag(TT) + rep(diag(TT), each = p) - 2 \* TT;

# Variance and correlation coefficients for Basic SparCC

rowT0 <- rowSums(T0);

var.w <- (rowT0 - sum(rowT0) / (2 \* p - 2))/(p - 2);

var.w[var.w < Vmin] <- Vmin;

#cor.w <- (outer(var.w, var.w, "+") - T0 ) /

# sqrt(outer(var.w, var.w, "\*")) / 2;

Is <- sqrt(1/var.w);

cor.w <- (var.w + rep(var.w, each = p) - T0) \* Is \* rep(Is, each = p) \* 0.5;

# Truncated correlation in [-1, 1]

cor.w[cor.w <= - 1] <- - 1;

cor.w[cor.w >= 1] <- 1;

# Left matrix of estimation equation

Lmat <- diag(rep(p - 2, p)) + 1;

# Remove pairs

rp <- NULL;

# Left components

cp <- rep(TRUE, p);

# Do loops until max iteration or only 3 components left

k <- 0;

while(k < kmax && sum(cp) > 3) {

# Left T0 = var(log(xi/xj)) after removing pairs

T02 <- T0;

# Store current correlation to find the strongest pair

curr\_cor.w <- cor.w;

# Remove diagonal

diag(curr\_cor.w) <- 0;

# Remove removed pairs

if(!is.null(rp)) {

curr\_cor.w[rp] <- 0;

}

# Find the strongest pair in vector form

n\_rp <- which.max(abs(curr\_cor.w));

# Remove the pair if geater than alpha

if(abs(curr\_cor.w[n\_rp]) >= alpha) {

# Which pair in matrix form

t\_id <- c(arrayInd(n\_rp, .dim = c(p, p)));

Lmat[t\_id, t\_id] <- Lmat[t\_id, t\_id] - 1;

# Update remove pairs

n\_rp <- c(n\_rp, (p + 1) \* sum(t\_id) - 2 \* p - n\_rp);

rp <- c(rp, n\_rp);

# Update T02

T02[rp] <- 0;

# Which component left

cp <- (diag(Lmat) > 0);

# Update variance and truncated lower by Vmin

var.w[cp] <- solve(Lmat[cp, cp], rowSums(T02[cp, cp]));

var.w[var.w <= Vmin] <- Vmin;

# Update correlation matrix and truncated by [-1, 1]

#cor.w <- (outer(var.w, var.w, "+") - T0 ) /

# sqrt(outer(var.w, var.w, "\*")) / 2;

Is <- sqrt(1/var.w);

cor.w <- (var.w + rep(var.w, each = p) - T0) \*

Is \* rep(Is, each = p) \* 0.5;

# Truncated correlation in [-1, 1]

cor.w[cor.w <= - 1] <- - 1;

cor.w[cor.w >= 1] <- 1;

}

else {

break;

}

#

k <- k + 1;

}

# Covariance

Is <- sqrt(var.w);

cov.w <- cor.w \* Is \* rep(Is, each = p);

#

return(list(cov.w = cov.w, cor.w = cor.w));

}

#---------------------------------------------------------------------------------------------------------------------

**CCLasso:**

#function -CCLasso correlation test#

########################################################################

# File: cclasso.R

# Aim : Correlation inference for compositional data through lasso

#---------------------------------------------------------------------------------------------------------------------

# Author : Fang Huaying (Peking University)

# Email : hyfang@pku.edu.cn

# Date : 2016-01-08

# Version: 2.0

#---------------------------------------------------------------------------------------------------------------------

# Main function: cclasso(x, counts = FALSE, pseudo = 0.5, k\_cv = 3,

# lam\_int = c(1e-4, 1), k\_max = 20, n\_boot = 20)

#

# Input:

# x ------ n x p data matrix (row/column is sample/variable)

# n samples & p compositional variables

# counts ------ Is the compositional data matrix a count matrix?

# Default: FALSE

# pseudo ------ pseudo count if counts = TRUE

# Default: 0.5

# k\_cv ------ folds of cross validation

# Default: 3

# lam\_int ------ tuning parameter interval

# Default: [1e-4, 1]

# k\_max ------ maximum iterations for golden section method

# Default: 20

# n\_boot ------ Bootstrap times

# Default: 20

# Output:

# A list structure contains:

# var\_w ------ variance estimation

# cor\_w ------ correlation estimation

# p\_vals ------ p-values for elements of cor\_w equal 0 or not

# lambda ------ final tuning parameter

# info\_cv ------ information for cross validation

#---------------------------------------------------------------------------------------------------------------------

cclasso <- function(x, counts = FALSE, pseudo = 0.5, k\_cv = 3,

lam\_int = c(1e-4, 1), k\_max = 20, n\_boot = 20) {

n <- nrow(x);

p <- ncol(x);

if(counts) {

x <- x + pseudo;

x <- x / rowSums(x);

}

x <- log(x);

vx2 <- stats::var(x);

# Diagonal weight for loss function

rmean\_vx2 <- rowMeans(vx2);

wd <- 1/diag(vx2 - rmean\_vx2 - rep(rmean\_vx2, each = p) + mean(rmean\_vx2));

wd2 <- sqrt(wd);

# Some global parameters for optimization with single lambda

rho <- 1;

u\_f <- eigen(diag(p) - 1/p)$vectors;

wd\_u <- (t(u\_f) %\*% (wd \* u\_f))[-p, -p];

wd\_u\_eig <- eigen(wd\_u);

d0\_wd <- 1 / ( (rep(wd\_u\_eig$values, each = p-1) + wd\_u\_eig$values) /

(2 \* rho) + 1 );

u0\_wd <- wd\_u\_eig$vectors;

# Golden section method for the selection of lambda (log10 scale)

sigma <- vx2;

lam\_int2 <- log10(range(lam\_int));

a1 <- lam\_int2[1];

b1 <- lam\_int2[2];

# Store lambda and corresponding cross validation's loss

lams <- NULL;

fvals <- NULL;

# Two trial points in first

a2 <- a1 + 0.382 \* (b1 - a1);

b2 <- a1 + 0.618 \* (b1 - a1);

fb2 <- cv\_loss\_cclasso(lambda2 = 10^b2 / rho, x = x, k\_cv = k\_cv,

sigma = sigma, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd,

wd2 = wd2);

lams <- c(lams, b2);

fvals <- c(fvals, fb2$cv\_loss);

fa2 <- cv\_loss\_cclasso(lambda2 = 10^a2 / rho, x = x, k\_cv = k\_cv,

sigma = fb2$sigma, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd,

wd2 = wd2);

lams <- c(lams, a2);

fvals <- c(fvals, fa2$cv\_loss);

# Error tolerance for convergence

err\_lam2 <- 1e-1 \* max(1, lam\_int2);

err\_fval <- 1e-4;

err <- b1 - a1;

k <- 0;

while(err > err\_lam2 && k < k\_max) {

fval\_max <- max(fa2$cv\_loss, fb2$cv\_loss);

if(fa2$cv\_loss > fb2$cv\_loss) {

a1 <- a2;

a2 <- b2;

fa2 <- fb2;

b2 <- a1 + 0.618 \* (b1 - a1);

fb2 <- cv\_loss\_cclasso(lambda2 = 10^b2 / rho, x = x, k\_cv = k\_cv,

sigma = fa2$sigma, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd,

wd2 = wd2);

lams <- c(lams, b2);

fvals <- c(fvals, fb2$cv\_loss);

} else {

b1 <- b2;

b2 <- a2;

fb2 <- fa2;

a2 <- a1 + 0.382 \* (b1 - a1);

fa2 <- cv\_loss\_cclasso(lambda2 = 10^a2 / rho, x = x, k\_cv = k\_cv,

sigma = fb2$sigma, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd,

wd2 = wd2);

lams <- c(lams, a2);

fvals <- c(fvals, fa2$cv\_loss);

}

fval\_min <- min(fa2$cv\_loss, fb2$cv\_loss);

k <- k + 1;

err <- b1 - a1;

if(abs(fval\_max - fval\_min) / (1 + fval\_min) <= err\_fval) {

break;

}

}

info\_cv <- list(lams = lams, fvals = fvals, k = k + 2,

lam\_int = 10^c(a1, b1));

if(a1 == lam\_int2[1] || b1 == lam\_int2[2]) {

cat("WARNING:\n", "\tOptimal lambda is near boundary! ([", 10^a1, ",",

10^b1, "])\n", sep = "");

}

lambda <- 10^((a2 + b2)/2);

# Bootstrap for cclasso

lambda2 <- lambda / rho;

info\_boot <- boot\_cclasso(x = x, sigma = fb2$sigma, lambda2 = lambda2,

n\_boot = n\_boot, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd);

return(list(var\_w = info\_boot$var\_w, cor\_w = info\_boot$cor\_w,

p\_vals = info\_boot$p\_vals, lambda = lambda, info\_cv = info\_cv));

}

#---------------------------------------------------------------------------------------------------------------------

# Bootstrap for cclasso

boot\_cclasso <- function(x, sigma, lambda2, n\_boot = 20,

wd, u\_f, u0\_wd, d0\_wd) {

n <- nrow(x);

p <- ncol(x);

# Store the result of bootstrap

cors\_boot <- matrix(0, nrow = p \* (p - 1)/2, ncol = n\_boot + 1);

vars\_boot <- matrix(0, nrow = p, ncol = n\_boot + 1);

cors\_mat <- matrix(0, p, p);

ind\_low <- lower.tri(cors\_mat);

# Bootstrap procedure

sam\_boot <- matrix(sample(1:n, size = n \* n\_boot, replace = T),

ncol = n\_boot);

for(k in 1:n\_boot) {

ind\_samp <- sam\_boot[, k];

sigma2 <- cclasso\_sub(sigma = sigma, vx = var(x[ind\_samp, ]),

lambda2 = lambda2, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd);

vars\_boot[, k] <- diag(sigma2);

Is <- 1 / sqrt(vars\_boot[, k]);

cors\_mat <- Is \* sigma2 \* rep(Is, each = p);

cors\_boot[, k] <- cors\_mat[ind\_low];

}

Is <- 1 / sqrt(diag(sigma));

cors\_mat <- sigma \* Is \* rep(Is, each = p);

cors\_boot[, n\_boot + 1] <- cors\_mat[ind\_low];

vars\_boot[, n\_boot + 1] <- diag(sigma);

#------------------------------------------------------------------------------------------------------------------

# Variance estimation via bootstrap

vars2 <- rowMeans(vars\_boot);

#------------------------------------------------------------------------------------------------------------------

# Correlations' relationship for artificial null sample

tol\_cor <- 1e-3;

sam\_art0 <- matrix(rnorm(n \* p), nrow = n) \* rep(sqrt(vars2), each = n);

cors\_art0 <- cor(sam\_art0)[ind\_low];

sam\_art <- sam\_art0 - log(rowSums(exp(sam\_art0)));

sigma\_art <- cclasso\_sub(sigma = sigma, vx = var(sam\_art),

lambda2 = lambda2, wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd);

Is <- 1 / sqrt(diag(sigma\_art));

cors\_mat <- Is \* sigma\_art \* rep(Is, each = p);

cors\_art2 <- cors\_mat[ind\_low];

# Bias of estimation between absolute data and cclasso of compositional data

cors0m2 <- log( ((1 + cors\_art0) \* (1 - cors\_art2)) / ((1 + cors\_art2) \*

(1 - cors\_art0)) );

tmp <- abs(cors\_art2) >= tol\_cor;

bias02 <- ifelse(sum(tmp), median(abs(cors0m2)[tmp]), 0);

# Modification of estimation for cclasso

cors2 <- log( (1 + cors\_boot) / (1 - cors\_boot) );

cors2mod <- (cors\_boot >= tol\_cor) \* (cors2 + bias02) +

(cors\_boot <= -tol\_cor) \* (cors2 - bias02);

cors2mod <- 1 - rowMeans(2 / (exp(cors2mod) + 1));

cors2\_mat <- diag(p);

cors2\_mat[ind\_low] <- cors2mod;

cors2\_mat <- t(cors2\_mat);

cors2\_mat[ind\_low] <- cors2mod;

# P-values with null distribution of correlation estimations of absolute data

p\_vals <- pt(cors2mod \* sqrt((n - 2) / (1 - cors2mod^2)), df = n - 2);

p\_vals <- ifelse(p\_vals <= 0.5, p\_vals, 1 - p\_vals);

pval\_mat <- diag(p);

pval\_mat[ind\_low] <- p\_vals;

pval\_mat <- t(pval\_mat);

pval\_mat[ind\_low] <- p\_vals;

#---------------------------------------------------------------------------

return(list(var\_w = vars2, cor\_w = cors2\_mat, p\_vals = pval\_mat));

}

#-------------------------------------------------------------------------------

# cross validation's loss of cclasso for single lambda

cv\_loss\_cclasso <- function(lambda2, x, k\_cv, sigma,

wd, u\_f, u0\_wd, d0\_wd, wd2) {

n <- nrow(x);

p <- ncol(x);

n\_b <- floor(n / k\_cv);

cv\_loss <- 0;

for(k in 1:k\_cv) {

itest <- (n\_b \* (k - 1) + 1):(n\_b \* k);

vxk <- stats::var(x[itest, ]);

vx2k <- stats::var(x[-itest, ]);

sigma <- cclasso\_sub(sigma = sigma, vx = vx2k, lambda2 = lambda2,

wd = wd, u\_f = u\_f, u0\_wd = u0\_wd, d0\_wd = d0\_wd);

dsig <- sigma - vxk;

tmp <- rowMeans(dsig);

dsig <- dsig - tmp - rep(tmp, each = p) + mean(tmp);

cv\_loss <- cv\_loss + base::norm(wd2 \* dsig, "F")^2;

}

return(list(cv\_loss = cv\_loss, sigma = sigma));

}

#-------------------------------------------------------------------------------

# cclasso for single lambda

cclasso\_sub <- function(sigma, vx, lambda2,

wd, u\_f, u0\_wd, d0\_wd,

k\_max = 200, x\_tol = 1e-4) {

p <- ncol(sigma);

sigma2 <- sigma;

LAMBDA <- matrix(0, p, p);

lambda2 <- matrix(lambda2, p, p);

diag(lambda2) <- 0;

k <- 0;

err <- 1;

while(err > x\_tol && k < k\_max) {

# Update sigma

x\_sigma <- t(u\_f) %\*% ((sigma2 - vx) - LAMBDA) %\*% u\_f;

x\_sigma[-p,-p] <- u0\_wd %\*% ((t(u0\_wd) %\*% x\_sigma[-p, -p] %\*% u0\_wd) \*

d0\_wd) %\*% t(u0\_wd);

sigma\_new <- vx + u\_f %\*% x\_sigma %\*% t(u\_f);

# Update sigma2

A <- LAMBDA + sigma\_new;

sigma2\_new <- (A > lambda2) \* (A - lambda2) + (A < -lambda2) \*

(A + lambda2);

# Update Lambda

LAMBDA <- LAMBDA + (sigma\_new - sigma2\_new);

err <- max( abs(sigma\_new - sigma)/(abs(sigma) + 1),

abs(sigma2\_new - sigma2)/(abs(sigma2) + 1) );

k <- k + 1;

sigma <- sigma\_new;

sigma2 <- sigma2\_new;

}

if(k >= k\_max) {

cat("WARNING of cclasso\_sub:\n", "\tMaximum Iteration:", k\_max,

"&& Relative error:", err, "!\n");

}

return(sigma);

}

#---------------------------------------------------------------------------------------------------------------------

# **cosine similarity**#

cosine=function(x,boot\_n=20)

{

#normalization#

x<-scale(x, center = TRUE, scale = TRUE)

#cosine value count#

x.row<-as.numeric(nrow(x))

x.col<-as.numeric(ncol(x))

bb <- matrix(rep(0,x.row^2),x.row,x.row)

for(j in 1:x.row)

{

for(k in 1:x.row)

{

if(j<k)

bb[j,k] = sum(t(x[j,])\*x[k,])/sqrt((sum(x[j,]^2))\*sum(x[k,]^2))

}

}

#set up the original p value#

p<-matrix(rep(0,x.row^2),x.row,x.row)

#for rountine test#

for(l in 1:boot\_n)

{

#bootstrap to generate shuffled datasets#

boot\_xC<-matrix(sample(x,replace = T),x.row,x.col)

#count the cosine of shuffled datasets#

bb1 <- matrix(rep(0,x.row^2),x.row,x.row)

for(m in 1:x.row)

{

for(n in 1:x.row)

{

if(m<n)

bb1[m,n]=sum(t(boot\_xC[m,])\*boot\_xC[n,])/sqrt((sum(boot\_xC[m,]^2))\*sum(boot\_xC[n,]^2))

}

}

#Pseudo p-value calculation#

tempC<-ifelse(bb1<=bb,1,0)

p<-tempC+p

}

p<-p/boot\_n

#get the right triangle data#

p[lower.tri(p)]=0

diag(p)=0

return(list(r.cosine=bb,p=p))

}