LDA-regression tutorial

Qing Zhao, Denis Valle, Ermias Azeria

Introduction

For this tutorial we will introduce how to use customized code to run LDA-regression model. We will illustrate the use of LDA-regression model with a simulation study, but we will also introduce how to run this model on real-world data.

Simulate data

First, let's determine some basic values that will used to simulate data. These values include the number of sites, number of species, number of observation opportunities per site and species, number of covariates, and number of communities. These values, except for the number of communities, are normally known for real-world study, while the number of communities can be tested with different model selection approaches such as WAIC or truncated stick-breaking prior. Note that we assume that the number of observation opportunities per site and species is always 5 here but in real-world study this number can vary across sites and species. Also, nx here is the number of covariates plus one because the first column of x will always be 1 for intercept. We then will simulate the covariates from normal distributions. We will also simulated the intercept and slope parameters of the multinomial regression in LDA-regression from a random uniform distribution from -1 to 1. We will also simulate theta's, phi's, and eventurally the observed number of presence of species y's.

```
nl <- 500 # number of sites
ns <- 200 # number of species
nm <- matrix(5, nl, ns) # number of observation opportunities per site and species
nx <- 5 # number of covariates plus one
nc <- 4 # number of communities

# Simulate the covariates
x <- cbind(1, matrix(rnorm(nl*(nx-1), 0, 1), nrow=nl, ncol=nx-1))

# Simulate the intercept and slope parameters
beta <- matrix(runif(nx*(nc-1), -1, 1), nrow=nx, ncol=nc-1)

# Simulate theta's, the proportions of groups at each site
mu <- x %*% beta
z <- matrix(rnorm(nl*(nc-1), mu, 1),nrow=nl,ncol=nc-1)
vmat <- cbind(exp(z),1)
theta <- vmat / rowSums(vmat)

# Simulate phi's, the proportion of species for each community
phi <- matrix(rbeta(nc*ns, .2, .8), nrow=nc, ncol=ns)

# Simulate y's, the number of detections for each species at each site
# during the nm observation opportunities.
pi <- theta %*% phi # probability of occurrence
y <- matrix(rbinom(nl*ns, nm, pi), nrow=nl, ncol=ns) # number of detections
```

Up until this point we have all the data we need for analysis. Note that no all of the things we generated so far are available for real-world data. For a real-world study, we will have the number of detections y, the

number of observation opportunities nm, and the covariates' matrix x. We can also easily get the number of sites nl and the number of species ns, which will be the number of rows and columns of y, respectively. We will also get nx as the number of columns of x. We will also need to determine the number of communities, which is normally unknown but can be tested with WAIC etc. Here we will assume that the number of communities is known. Things like theta's, phi's and beta's are unknown and are our interests of inference.

Import functions

Next we will import the functions we developed for MCMC computing. These functions are available on GitHub (https://github.com/QingZhaoUFL/LDA-regression). These are the main functions of this model, and it's highly recommended to look into thse codes to understand the details about this model. These functions can also be found at the end of this document.

```
source(paste(c(path, 'code/functions.R'), collapse=''))
```

Run LDA-regression

Next we will run the MCMC computing. First we will need to determine the number of iterations, burn-in, and adaptations. We then will determine initial values. The last, we will run MCMC computing.

```
niter <- 3000 # number of iterations
nburn <- round(niter * .8) # number of burn-in
adapt <- round(niter * .5) # number of adaptations

# Determine initial values
z.jump <- matrix(.05, nrow=nl, ncol=nc-1)
phi.jump <- matrix(.05, nrow=nc, ncol=ns)

param <- list()

param$z <- matrix(0,nl,nc-1)
param$theta <- matrix(1/nc,nl,nc)
param$phi <- matrix(0.5,nc,ns)
param$beta <- matrix(0,nx,nc-1)

theta.post <- array(, dim=c(nl, nc, niter))
theta.post[,,1] <- param$theta

z.jump.post <- z.accept <- array(, dim=c(nl, nc-1, niter))
z.jump.post[,,1] = z.jump
z.accept[,,1] <- FALSE

phi.post <- phi.jump.post <- phi.accept <- array(, dim=c(nc, ns, niter))
phi.post[,,1] <- param$phi
phi.jump.post[,,1] <- phi.jump
phi.accept[,,1] <- FALSE

beta.post <- array(, dim=c(nx, nc-1, niter))
beta.post[,,1] <- param$beta

# Pre-calculate some useful stuff
invT=diag(1,nx)
invT[1,1]=1/25
```

```
tx=t(x)
xtx=tx%*%x
var1=solve(xtx+invT)
s. <- svd(var1)
sqrtVar1=s.$u %*% diag(sqrt(s.$d))

# Run MCMC computing.
for (i in 2:niter) {
  tmp <- update.theta(param, jump=z.jump, nl, nc, y, x, nm)
  param$z = tmp$z
  theta.post[,,i] <- param$theta <- tmp$theta
  z.accept[,,i] <- tmp$accept

  tmp <- update.phi(param, jump=phi.jump, nc, ns, y, nm, a.phi=1, b.phi=1)
  phi.post[,,i] <- param$phi <- tmp$phi
  phi.accept[,,i] <- tmp$accept

  beta.tmp <- update.beta(param, nx, nc, tx, sqrtVar1, var1)
  beta.post[,,i] <- param$beta <- beta.tmp

  #tune jump parameter
  if (i < adapt & i%%100==0){
    tmp <- jumpTune(accept=z.accept, jump=z.jump, ni=i, low=.3, high=.8)
    z.jump.post[,,i] <- z.jump <- tmp

    tmp <- jumpTune(accept=phi.accept, jump=phi.jump, ni=i, low=.3, high=.8)
    phi.jump.post[,,i] <- phi.jump <- tmp
  }
}
```

Check the results

Note we have the posterior samples for theta's, phi's and beta's, as theta.post, phi.post, and beta.post, respectively. Here we will show you how to check the inference of beta's. One important thing we want to mention is that often there will be "community flipping" in the results, due to the fact that the model cannot guarantee the identified order of communities is the same as the real one. When community flipping occurrs (say community 1 is identifies as community 2), the corresponding beta's are flipped as well. Evenmore, since the beta's in multinomial regression represent the comparison of a given community with the last community, the beta's can have totally different values when a different community is identified as the last community. Therefore, it's not always meaningful to compare the estimates of beta's with their true values. Alternatively, we will need to construct response curves for such comparison. The response curves should remain the same even when community flipping occurrs. Also note that even if the last community has no beta's, it will have response curves. The following code shows how to construct response curves.

```
beta.est <- beta.post[,,(nburn+1):niter]

nl.res <- 100

par(mfrow=c(2,2))
par(mar=c(4,4,1,1))

for (i in 1:(nx-1)) {
```

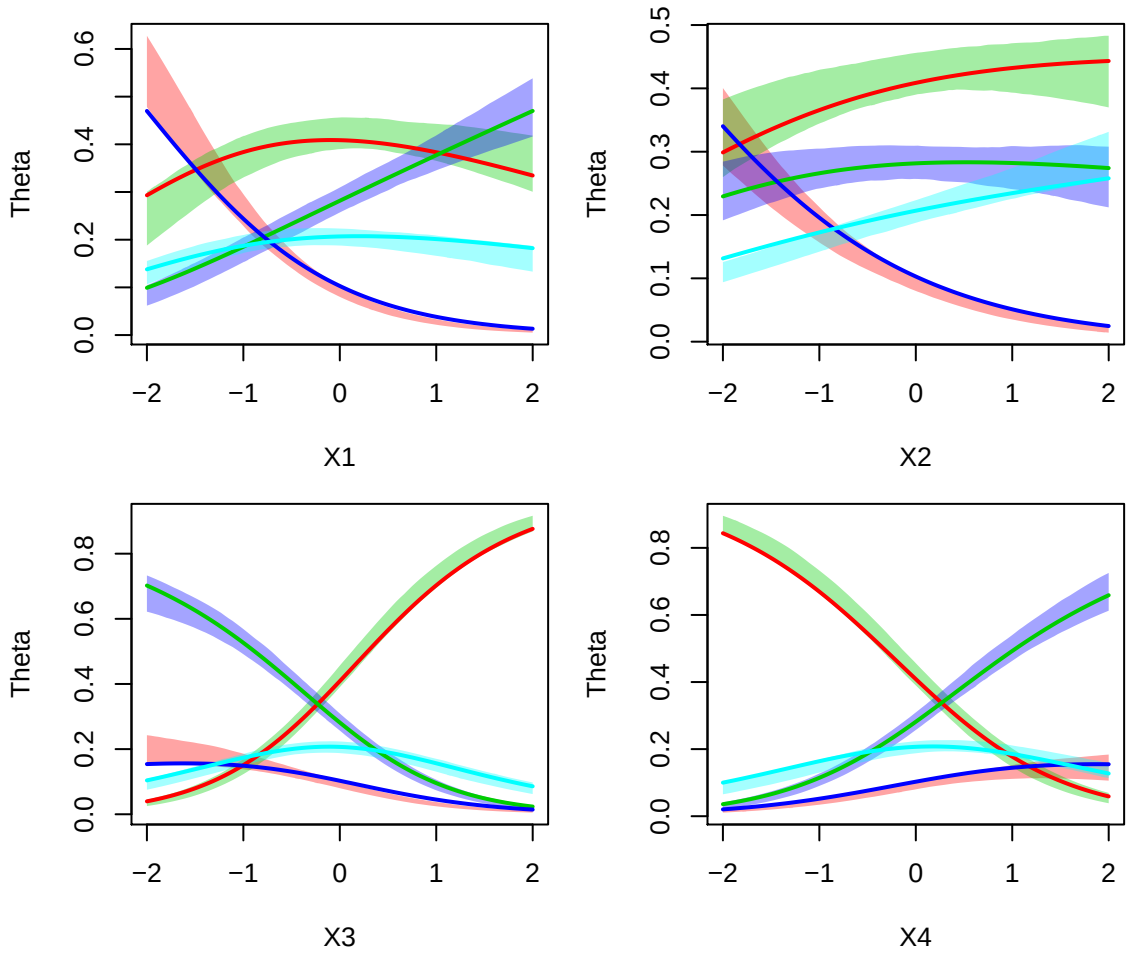
```
  x.res <- cbind(1, matrix(0, nl.res, nx-1))
  x.res[,i+1] <- seq(-2, 2, length.out=nl.res)

  mu.res.true <- x.res %*% beta
  vmat.res.true <- cbind(exp(mu.res.true), 1)
  theta.res.true <- vmat.res.true / rowSums(vmat.res.true)

  theta.res <- array(, dim=c(nl.res, nc, dim(beta.est)[3]))
  for (m in 1:dim(beta.est)[3]) {
    mu.res <- x.res %*% beta.est[,,m]
    vmat.res <- cbind(exp(mu.res), 1)
    theta.res[,,m] <- vmat.res / rowSums(vmat.res)
  }
  theta.res.qt <- apply(theta.res, 1:2, quantile, probs=c(.025, .975))

  plot(theta.res.true[,1] ~ x.res[,i+1], ylim=range(c(theta.res.true,theta.res.qt)),
       type='n', xlab=paste('X', i, sep=''), ylab='Theta')
  for (j in 1:nc) {
    rgb <- as.vector(col2rgb(j+1))
    col <- rgb(red=rgb[1], green=rgb[2], blue=rgb[3], alpha=255*.36, maxColorValue=255)
    polygon(x=c(x.res[,i+1],rev(x.res[,i+1])),
            y=c(theta.res.qt[1,,j],rev(theta.res.qt[2,,j])), border=NA, col=col)
  }
  for (j in 1:nc) {
    lines(theta.res.true[,j] ~ x.res[,i+1], col=j+1, lwd=2)
  }
} # i
```

The figure below shows a result for 4 communities and 4 covariates. Solid lines represent the true relationships, the bands represent the 95% CI of the estimated relationships, and different colors are for different communities. Mismatch of colors (say a red line and a green band) represent community flipping. We can see here that even when community flipping occurrs, the 95% CI of the estimated relationships can contain the true relationships.



Functions

Below are the functions used in MCMC computing.

```
#generates multivariate normal with mean 0 and Sigma=sqrtVar1%*%t(sqrtVar1)
rmvnorm1 <- function (sqrtVar1) {
  sqrtVar1 %*% rnorm(ncol(sqrtVar1))
} # rmvnorml

tnorm <- function(n, lo, hi, mu, sig) {
  # generates truncated normal variates based on
  # cumulative normal distribution normal truncated lo and hi

  if (length(lo) == 1 & length(mu) > 1)
    lo <- rep(lo, length(mu))
  if (length(hi) == 1 & length(mu) > 1)
    hi <- rep(hi, length(mu))

  q1 <- pnorm(lo, mu, sig)  #cumulative distribution
  q2 <- pnorm(hi, mu, sig)  #cumulative distribution

  z <- runif(n, q1, q2)
```

```
  z <- qnorm(z, mu, sig)
  z[z == -Inf] <- lo[z == -Inf]
  z[z == Inf] <- hi[z == Inf]
  z
} # tnorm

fix.MH <- function(lo, hi, old1, new1, jump) {
  jold <- pnorm(hi, mean = old1, sd = jump) - pnorm(lo, mean = old1, sd = jump)
  jnew <- pnorm(hi, mean = new1, sd = jump) - pnorm(lo, mean = new1, sd = jump)
  log(jold) - log(jnew)  #add this to pnew
} # fix.MH

get.logl <- function(theta, phi, y, nm) {
  prob <- theta %*% phi
  cond <- prob < 1e-05
  prob[cond] <- 1e-05
  cond <- prob > 0.99999
  prob[cond] <- 0.99999
  dbinom(y, size = nm, prob = prob, log = T)
} # get.logl

acceptMH <- function(p0, p1, x0, x1, BLOCK) {
  # accept for M, M-H if BLOCK, then accept as a block, otherwise, accept individually

  nz <- length(x0)  #no. to accept
  if (BLOCK)
    nz <- 1

  a <- exp(p1 - p0)  #acceptance PR
  z <- runif(nz, 0, 1)
  keep <- which(z < a)

  if (BLOCK & length(keep) > 0)
    x0 <- x1
  if (!BLOCK)
    x0[keep] <- x1[keep]
  accept <- length(keep)

  list(x = x0, accept = accept)
} # acceptMH

update.theta <- function(param, jump, nl, nc, y, x, nm) {
  phi <- param$phi
  beta <- param$beta
  mu <- x %*% beta

  z.ori <- z.old <- param$z
  vmat.old=cbind(exp(z.old),1)

  z.tmp = rnorm(nl*(nc-1), mean=z.old, sd=jump)
  z.proposed = matrix(z.tmp,nrow=nl,ncol=nc-1)
  vmat.proposed <- cbind(exp(z.proposed),1)
```

```
  for (j in 1:(nc-1)) {
    # last column has to be 1
    vmat.new <- vmat.old
    vmat.new[,j] <- vmat.proposed[,j]

    theta.old <- vmat.old / rowSums(vmat.old)
    theta.new <- vmat.new / rowSums(vmat.new)

    prob.old <- get.logl(theta=theta.old, phi=phi, y=y, nm=nm)
    prob.new <- get.logl(theta=theta.new, phi=phi, y=y, nm=nm)

    pold <- rowSums(prob.old) + dnorm(log(vmat.old[,j]), mu[,j], sigma, log = T)
    pnew <- rowSums(prob.new) + dnorm(log(vmat.new[,j]), mu[,j], sigma, log = T)

    k <- acceptMH(p0=pold, p1=pnew, x0=vmat.old[,j], x1=vmat.new[,j], BLOCK=F)
    vmat.old[,j] <- k$x
  }

  vmat <- vmat.old
  theta <- vmat / rowSums(vmat)
  z = log(vmat)[,-nc]
  list(theta=theta, z=z, accept=z.ori!=log(vmat.old[,-nc]))
} # update.theta

update.phi <- function(param, jump, nc, ns, y, nm, a.phi, b.phi) {
  theta <- param$theta

  phi.ori <- phi.old <- param$phi
  proposed <- matrix(tnorm(nc*ns, lo=0, hi=1, mu=phi.old, sig=jump), nc, ns)
  adj <- fix.MH(lo=0, hi=1, old1=phi.old, new1=proposed, jump=jump)

  for (j in 1:nc) {
    phi.new <- phi.old
    phi.new[j,] <- proposed[j,]

    prob.old <- get.logl(theta=theta, phi=phi.old, y=y, nm=nm)
    prob.new <- get.logl(theta=theta, phi=phi.new, y=y, nm=nm)

    pold <- colSums(prob.old) + dbeta(phi.old[j,], a.phi, b.phi, log=T)
    pnew <- colSums(prob.new) + dbeta(phi.new[j,], a.phi, b.phi, log=T)

    k <- acceptMH(p0=pold, p1=pnew + adj[j,], x0=phi.old[j,], x1=phi.new[j,], BLOCK=F)
    phi.old[j,] <- k$x
  }
  phi <- phi.old
  list(phi=phi, accept=phi.ori!=phi.old)
} # update.phi

update.beta <- function(param, nx, nc, tx, sqrtVar1, var1) {
  z <- param$z

  beta=matrix(, nx, nc-1)
  for (i in 1:(nc-1)){
```

```
    pmean <- tx%*%z[,i]
    beta[,i] <- rmvnorm1(sqrtVar1) + var1%*%pmean
  }
  beta
} # update.beta

jumpTune <- function(accept, jump, ni, low=.3, high=.8) {
  nstart <- ifelse(ni>=100, ni-99, 1)
  accept.rate <- apply(accept[,,nstart:ni], 1:2, mean)
  jump[accept.rate < low]  <- jump[accept.rate < low] * 0.5
  jump[accept.rate > high] <- jump[accept.rate > high] * 2
  jump
} # jumpTune
```