LDA-regression tutorial

Qing Zhao, Denis Valle, Ermias Azeria

Introduction

For this tutorial we will describe how to run the LDA-regression model using our customized code. To this end, we will rely on simulated data to illustrate the use of this model.

Simulate data

First, let's determine some basic values that will be used to simulate data, including 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 generally known for any given dataset. The number of communities can be determined with different model selection approaches such as WAIC or the truncated stick-breaking prior.

In this simulated dataset, we set the number of observation opportunities per site and species *nm* to 5 (in a real-world study, this number can vary across sites and species), the number of sites *nl* to 500, the number of species *ns* to 200, the number of communities *nc* to 4, and the number of covariates *nx-1* to 4. We then simulate the covariates from normal distributions and the intercept and slope parameters of the multinomial regression from uniform distributions from -1 to 1. Finally, we simulate theta's, phi's, and eventurally the observed number of presences for each species and each site *y*.

```
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
```

As mentioned above, in a real-world study, we would need to determine the number of communities. However, here we will assume that the number of communities is known and focus on estimating the sets of parameters theta, phi and beta.

Import functions

Next we will import the functions required for our Gibbs sampler. These are the main functions of this model, and we highly recommend users to study this code to understand how this model works. These functions are available on GitHub (https://github.com/QingZhaoUFL/LDA-regression) and can also be found at the end of this document.

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

Run LDA-regression

To run our Gibbs sampler, we first need to determine the overall number of iterations *niter* and the number of iterations for burn-in *nburn* and adaptation *adapt*. We then provide the initial values for the jump parameters (to be used in our Metropolis within Gibbs algorithms) and for the rest of the parameters being estimated. Finally, we run the Gibbs sampler.

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

# Determine jump parameters for Metropolis-Hastings
z.jump <- matrix(.05, nrow=nl, ncol=nc-1)
phi.jump <- matrix(.05, nrow=nc, ncol=ns)

# Determine initial values for parameters to be estimated
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)

# Set arrays to store the results from each iteration of the Gibbs sampler
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 quantities that are repeatedly used within the sampler
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 Gibbs sampler.
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
  }
}
```

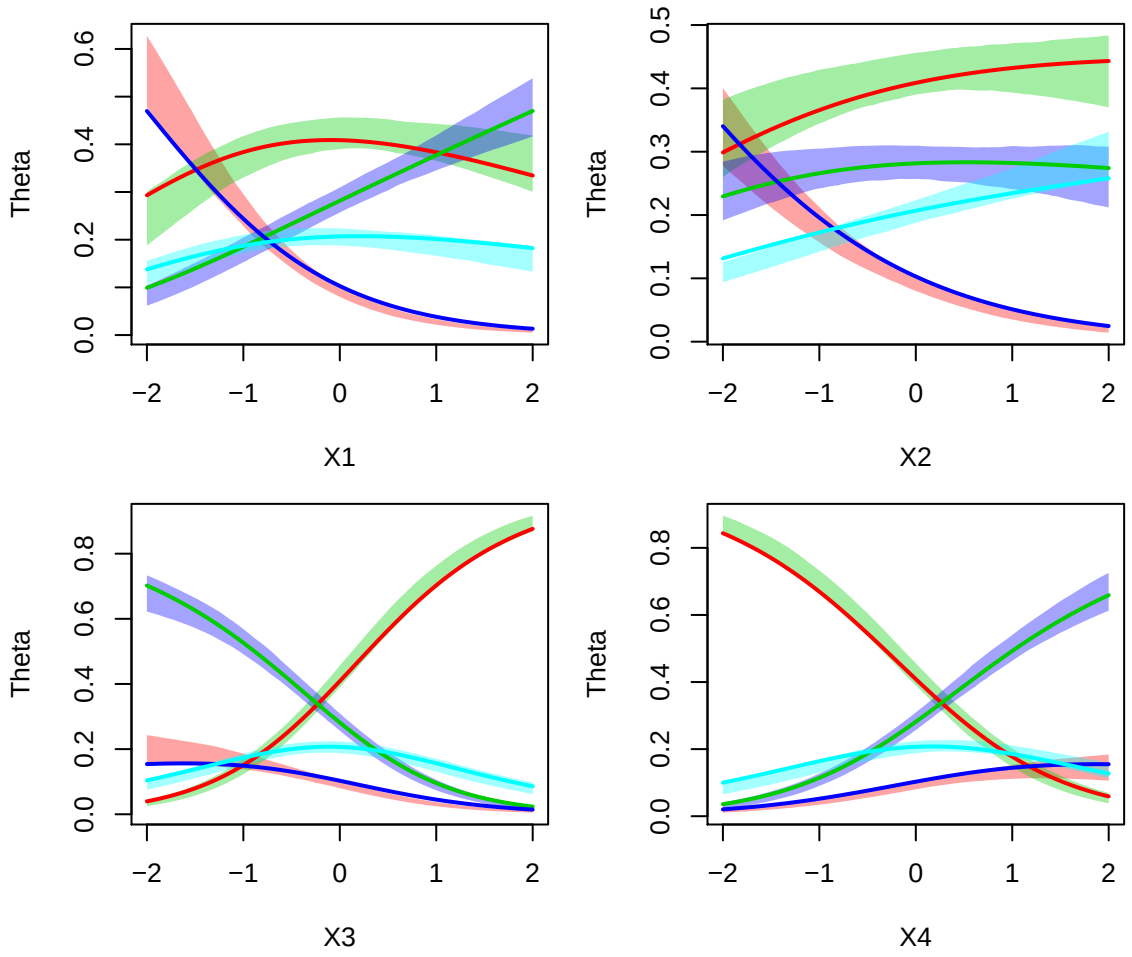
Check the results

Note that theta.post, phi.post, and beta.post hold posterior samples for theta, phi, and beta, respectively. Here we show how to check the inference on the beta parameters. One important observation is that the model cannot necessarily identify the correct order of these communities. When community flipping occurs (say community 1 is identifies as community 2), the corresponding beta's are flipped as well. Importantly, since the beta's in multinomial regression represent the comparison of a given community with the last community, these parameters can have completely different values when a different community is identified as the last community. Therefore, it's not always meaningful to compare the estimated beta's with their true values. Instead of comparing beta's, we construct response curves because these curves should remain the same even when community flipping occurs. Finally, note that even if the last community has no associated beta parameters, it will still have response curves. The following code shows how to create these 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 each covariate, calculate the corresponding response curve
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)
```

```
  #calculate the true response curves
  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)

  #calculate the estimated response curves
  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))

  #compare the true and the estimated response curves
  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 the results for 4 communities and 4 covariates. Solid lines represent the true relationships, the bands represent the 95% credible intervals (CI) of the estimated relationships, and different colors represent different communities. Mismatch of colors (e.g., 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 response curve still contains the true curve.



Functions

Below are the functions used for our Gibbs sampler.

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

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

  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

#calculates factor that needs to be included in Metropolis algorithm
#to correct for truncation in proposal distribution
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

#calculates the loglikelihood of the data
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

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

  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

#function to sample from the full conditional distribution of the theta's
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

#function to sample from the full conditional distribution of the phi's
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
```

```r
#function to sample from the full conditional distribution of the betas's
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

#updates the jump parameters used for our Metropolis within Gibbs algorithm
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
```