AMEN tutorial

Juan Tellez

2/27/2020

Introduction

This tutorial walks you through using the `AMEN` package with data on interstate conflict (MIDs) from Reiter and Stam (2003).

```
library(tidyverse)
library(haven)
```

Let's load the data below and look at it.

```
## read data and add variables
data = read_dta('/Users/JuamnTellez/Dropbox/papers/netsMatter/replications/Reiter_Stam_2003/input/doubl
head(data)
```

```
## # A tibble: 6 x 36
##   dummy  demb  dema  ally contig loglsrat majpow dispyrs dspline1 dspline2
##   <dbl> <dbl> <dbl> <dbl>  <dbl>    <dbl>  <dbl>   <dbl>    <dbl>    <dbl>
## 1     1    10    10     1      1     3.04      1       0    0        0
## 2     0    10    10     1      1     3.04      1       0    0        0
## 3     1    10    10     1      1     3.03      1       1   -0.872   -0.702
## 4     0    10    10     1      1     3.03      1       1   -0.872   -0.702
## 5     1    10    10     1      1     3.01      1       2   -6.98    -5.62
## 6     0    10    10     1      1     3.01      1       2   -6.98    -5.62
## # ... with 26 more variables: dspline3 <dbl>, personal <dbl>, single <dbl>,
## #   military <dbl>, democ <dbl>, notaiwan <dbl>, year <dbl>, stateb <dbl>,
## #   statea <dbl>, advanced <dbl>, dispute <dbl>, dictator <dbl>, persdem <dbl>,
## #   sideaa <dbl>, pdemdin <dbl>, id <dbl>, pdemdtar <dbl>, idyr <dbl>,
## #   singlea <dbl>, singleb <dbl>, mila <dbl>, milb <dbl>, sdemin <dbl>,
## #   sdemtar <dbl>, mdemin <dbl>, mdemtar <dbl>
```

For this tutorial, we only want the subset of the variables that were in the original analysis, and then we want to drop missing observations.

```
## get variables needed
base_vars = c('sideaa', "pdemdtar", "pdemdin", "personal", "military",
              "single", "democ", "contig", "majpow", "ally", "loglsrat",
              "advanced", "dispyrs", "dspline1", "dspline2", "dspline3",
              "statea", "stateb", "year")

## subset the data by variables needed and years
dataComp = na.omit(data[, base_vars])
```

Formatting the Data

Using the AMEN function requires formatting data into a particular structure. The primary distinction in data formatting is whether the outcome of interest represents a directed or undirected network.

If undirected, the AMEN function has three main inputs:

- Y: a T length **list** of $n \times n$ adjacency matrices, where T = number of years in the dataset and n = number of nodes in the network.
  - An adjacency matrix describes relationships between nodes in a particular year of data. For example, in an adjacency matrix of interstate MIDs, row i and column j takes a value of '1' if country i and country j had a MID in that year, and '0' otherwise. The diagonal in the adjacency matrix is typically missing. Y is a list of these adjacency matrices for the outcome variable, where each element in the list is a different year of data.
- Xdyad: a T length **list** of $n \times n \times p$ arrays, where p = number of dyadic covariates in dataset.
  - An array is a data object in R that contains a series of 'stacked' matrices for each year of data. An array of dimension (2, 3, 4), for example, contains 4 rectangular matrices each with 2 rows and 3 columns. Each matrix in the array describes the relationship between nodes with respect to some covariate. For example, for interstate alliances, row i and column j takes a value of '1' if country i and country j had an alliance in that year, and '0' otherwise. An array contains a matrix of this kind for each covariate going into the model. Xdyad is a list of these arrays, where each element in the list is a different year of data.
- Xrow: a T length **list** of $n \times p$ matrices, where p = number of monadic (nodal) covariates in dataset.
  - Each matrix in Xrow has the nodes running down the rows and the covariates in the dataset running along the columns. An entry in the matrix captures the value that a node takes on for each covariate in a given year. For example, if column j is GDP per capita and column k is population, then row i, column j measures country i's GDP per capita and row i, column k measures country i's population in a given year. Xrow is a list of these matrices, where each element in the list is a different year of data.

If directed, AMEN further requires:

- Xrow: a T length list of $n \times p$ matrices, where p = number of sender (nodal) covariates in dataset.
  - Xrow here is nearly identical to the undirected case. The only difference is that rather than each matrix containing covariate data on all nodes, in the directed context each matrix only contains data on the nodes that are acting as senders in that particular year. Using interstate MIDs as an example, if in 1983 only two countries initiated MIDs, Xrow would only contain data for those two countries (i.e., Xrow would only have two rows).
- Xcol: a T length list of $n \times p$ matrices, where p = number of receiver (nodal) covariates in dataset.
  - Xcol here is nearly identical to Xrow, except each matrix contains data on receiver nodes, not sender nodes. Using interstate MIDs as an example, if in 1983 only two countries had MIDs initiated against them, Xcol would only contain data for those two countries (i.e., Xcol would only have two rows).

The Reiter and Stam (2003) example is undirected.

Helper Functions

We provide some helper functions to make formatting easier.

First we need a *vector* of the years in the dataset, and then a *list* that tells us, *for each year*, which countries are present in the data (here represented by COW codes).

```
yrs = sort(unique(dataComp$year))
cntriesT = lapply(yrs, function(t){ as.character( unique( dataComp$statea[dataComp$year==t] ) ) })
```

Format Y (The Outcome)

Now we will format `Y`, the outcome variable, which is a list of adjacency matrices as described above. Roughly, the code below does the following: for each year of data, subset to the N countries present in that year, and

then recast the data frame into an $N \times N$ adjacency matrix, before finally storing it as an element into a list. Notice that Y is a list with 47 elements (one for each year in the data).

```
## format dependent variable
yVar = 'sideaa' # name of outcome in dataset
yList = lapply(1:length(yrs), function(ii){
  slice = data[ which(
    data$year==yrs[ii] &
      data$statea %in% cntriesT[[ii]] &
      data$stateb %in% cntriesT[[ii]]
  ), c('statea', 'stateb', yVar) ]
  adj = reshape2::acast(slice, statea ~ stateb, value.var=yVar)
  return( adj[ cntriesT[[ii]], cntriesT[[ii]] ] )
})
```

Here's an example of one year of data (only 5 countries shown for space purposes):

```
yList[[1]][1:5, 1:5]
```

```
##    20  2 40 41 42
## 20 NA  0  0  0  0
## 2   0 NA  0  0  0
## 40  0  0 NA  0  0
## 41  0  0  0 NA  0
## 42  0  0  0  0 NA
```

Format Xdyad, the Dyadic Covariates

Now we will format `Xdyad` from above, which includes the dyadic covariates. `Xdyad` is a T length **list** of $n \times n \times p$ arrays, where p = number of dyadic covariates in dataset.

Roughly, the code below does the following: for each year in the dataset, subset down to the countries present in that year and all of the dyadic variables in `dyad_vars`; then cast the data into an array that is $n \times n \times p$, before storing it into the list.

```
# get dyadic variables
dyad_vars = c('personal', 'military', 'single', 'democ', 'contig',
              'ally', 'majpow', 'loglsrat', 'advanced',
              'dispyrs', 'dspline1', 'dspline2', 'dspline3',
              'pdemdtar', 'pdemdin')

## format dyadic covariates
xDyadList = lapply(1:length(yrs), function(ii){
  slice = data[ which(
    data$year==yrs[ii] &
      data$statea %in% cntriesT[[ii]] &
      data$stateb %in% cntriesT[[ii]]
  ), c('statea', 'stateb', dyad_vars) ]
  sliceL = reshape2::melt(slice, id=c('statea','stateb'))
  adj = reshape2::acast(sliceL, statea ~ stateb ~ variable, value.var='value')
  return( adj[ cntriesT[[ii]], cntriesT[[ii]],  ] )
})
```

Here's an example of what the first year of data looks like for the first covariate, named `personal` (only 5 countries shown for space purposes):

```
xDyadList[[1]][1:5,1:5,1]
```

```
##    20  2 40 41 42
## 20 NA  0  0  0  0
## 2   0 NA  0  0  0
## 40  0  0 NA  0  0
## 41  0  0  0 NA  0
## 42  0  0  0  0 NA
```

We are done formatting the data and are ready for fitting the model.

Running AMEN

Beyond the data inputs, the AMEN function requires additional specification:

- model: how to model the outcome variable, e.g., 'logit'
- symmetric: whether the input network is symmetric
- intercept: whether to estimate an intercept
- nscan: number of iterations of the Markov chain
- burn: burn-in period
- odens: thinning interval
- R: dimension of the multiplicative effect (referred to as K in the paper)
- gof: whether to calculate goodness of fit statistics

There is often little theoretical reason to choose a particular value of R (above). One strategy is to estimate models at different values of R and compare goodness of fit statistics across models.

Given the computational intensity needed for parameter estimates to converge, parallelization strategies are recommended to speed up analysis. In addition, providing AMEN function with starting values, either dictated by theory, previous research, or previous runs can also help speed up convergence time.

Below, we load the library, and in `resultsPath` we set a path to store results.

```
## load most updated amen package
devtools::install_github('s7minhas/amen') ; library(amen)

## Run amen in parallel
library(doParallel) ; library(foreach)

## setup result output directory
resultsPath = '/Users/JuamnTellez/Dropbox/papers/netsMatter/replications/Reiter_Stam_2003/output/'
```

Now we set the parameters described above. Note that `latDims` here is the same as R above, and we have set it to 4.

```
## running in parallel varying k
imps = 10000  # number of interations in MCMC
brn =  2500   # number of burn-in
ods = 1 # thinning parameter
## (1 is suggested unless there is some evidence of autocorrelation in the chain)
latDims = 0:3 # dimension of latent space (search up to 3 dimension  is suggested)
seed=6886 # set your model seed
```

Now we run the model, in parallel. We fit a separate model for each element in `latDims` (so a model with a zero dimensional latent space, 1 dimensional latent space, etc.).

The key function here is `ame_repL` which fits the model. We wrap the whole function in a `foreach` call so we can run separate models for each element in `latDims`.

```
## select number of cores to locate
cl=makeCluster(4) ; registerDoParallel(cl) # number of clusters depends
## on how many cores your computer or cluster has

foreach(ii=1:length(latDims), .packages=c("amen")) %dopar% {

  ameFit = ame_repL(
    Y=yList,Xdyad=xDyadList,
    Xrow=NULL,Xcol=NULL,
    model="bin",symmetric=FALSE,intercept=TRUE,R=latDims[ii],
    nscan=imps, seed=seed, burn=brn, odens=ods,
    plot=FALSE, print=FALSE, gof=TRUE,
    periodicSave=TRUE)

  save(ameFit, file=paste0(resultsPath, 'model_k', latDims[ii],'_v1.rda') ) # save the output of your d
}

## terminate parallel sockets
stopCluster(cl)
```

In cases when the model fails to converge you may want to run the model again or with a longer chain. In this case, it is often useful to set starting values which can help with convergence issues.

Below, we demonstrate how to take starting values from a prior run and then fit the model again.

```
## load resuls from previous model run if needed
prevModelFiles = paste0(resultsPath, 'model_k', latDims,'_v1.rda')




## select number of cores to locate
cl=makeCluster(4) ; registerDoParallel(cl)

foreach(ii=1:length(latDims), .packages=c("amen")) %dopar% {

  load(prevModelFiles[ii]) # load previous model run

  startVals0 = ameFit$'startVals' # extract start vals

  rm(ameFit)  # dump the rest of the model run

  ameFit = ame_repL(
    Y=yList,Xdyad=xDyadList,
    Xrow=NULL,Xcol=NULL,
    model="bin",symmetric=FALSE,intercept=TRUE,R=latDims[ii],
    nscan=imps, seed=seed, burn=brn, odens=ods,
    plot=FALSE, print=FALSE, gof=TRUE, startVals=startVals0,
    periodicSave=TRUE)

  # save the output of your data
  save(ameFit, file=paste0(resultsPath, 'model_k', latDims[ii],'_v2.rda') )
}
```

```
## terminate parallel sockets
stopCluster(cl)
```

Posterior Analysis

Once we are satisfied that the model has converged, we can do posterior analysis to examine our parameter estimates. Below we load the results from the `AMEN` run. Simplicity, we only load results from $R = 0$.

```
## load model fits from 0 to 3 dimensions
load( paste0(resultsPath,'ameFit_k0.rda') ) ; ameFit_k0=ameFit
# load( paste0(resultsPath,'ameFit_k1.rda') ) ; ameFit_k1=ameFit
# load( paste0(resultsPath,'ameFit_k2.rda') ) ; ameFit_k2=ameFit
# load( paste0(resultsPath,'ameFit_k3.rda') ) ; ameFit_k3=ameFit
```

We can look at goodness of fit (GOF) statistics with the following:

```
## check out goodness of fit stats in posterior estimate for each dimension
amen::gofPlot(ameFit_k0$GOF, symmetric = TRUE)
```

We can also summarize the distribution of our beta parameters, below, using the `summStats` function which returns the mean, median, standard deviation, 95%, and 99% credible intervals:

```
## check out beta parameters (posterior estimates for coefficients)
## and convergence for each dimension
summStats = function(x){
  # credible intervals
  res=c(mu=mean(x),med=median(x),sd=sd(x),quantile(x,probs=c(0.025,0.05,0.95,0.975)))
  round(res, 3)
}
apply(ameFit_k0$BETA, 2, summStats) %>% t() %>% nrow()
```

We can also verify that parameters have converged for sender and receiver effects with the code below:

```
betaIndices<-split(1:ncol(ameFit_k0$BETA), ceiling(seq_along(1:ncol(ameFit_k0$BETA))/5))
```

And finally make plots to assess the converge of parameters:

```
## plot the parameters convergence using dimension k=0
for(bIndex in betaIndices){ amen::paramPlot( ameFit_k0$BETA[,bIndex,drop=FALSE] ) }
amen::paramPlot(ameFit_k0$VC)
```