Generative Network modelling examples

2024-02-06

Introduction

This document contains code for the modelling examples included in the paper 'An introduction to generative network models and how they may be used to study animal sociality' by Brask et al.

Set-up

To run these examples, you'll need to first install and then load several R libraries.

Some of the examples use the okaapibeta package (the beta version of the okaapi package), which generates networks based on the trait preference model (see the list of tools for generative network modelling in main text). This package is available at Github and can be installed by the following procedure: first install the remotes package (only necessary if you don't have this or the devtools package installed already), then load that package (to get access to the install_github function), then run the following code in R: install_github("bohrbrask/okaapibeta").

Modelling examples

EXAMPLE I: Emergence of animal social network structures

We want to investigate how generative processes - here social preferences - affect network structure. Specifically, We investigate how social preferences for individuals similar to oneself (homophily) affects social network structure in a social system where there are five matrilines and individuals prefer to socialize with those of their own matriline.

To do this, the code makes two ensembles of networks, one with social preferences and one without (i.e. random structure), measures network metrics on them, and plots the metrics for comparison. The networks are generated with the trait preference model, which is available in the okaapibeta package (see the Set-up section above for information on this package).

The two network types are generated with the same settings, apart from the importance of preferences (which are zero for the random network).

Network generation parameters

First we set parameters which we will use in the network generation:

```
n <- 100                                    # number of nodes (individuals) in each network
k <- 10                                     # average degree (average number of links each indivi
traittypes <- 'cate'                        # trait type. We use a categorical trait to simulate
allncats <- 5                               # trait category number. We use five categories (matr
```

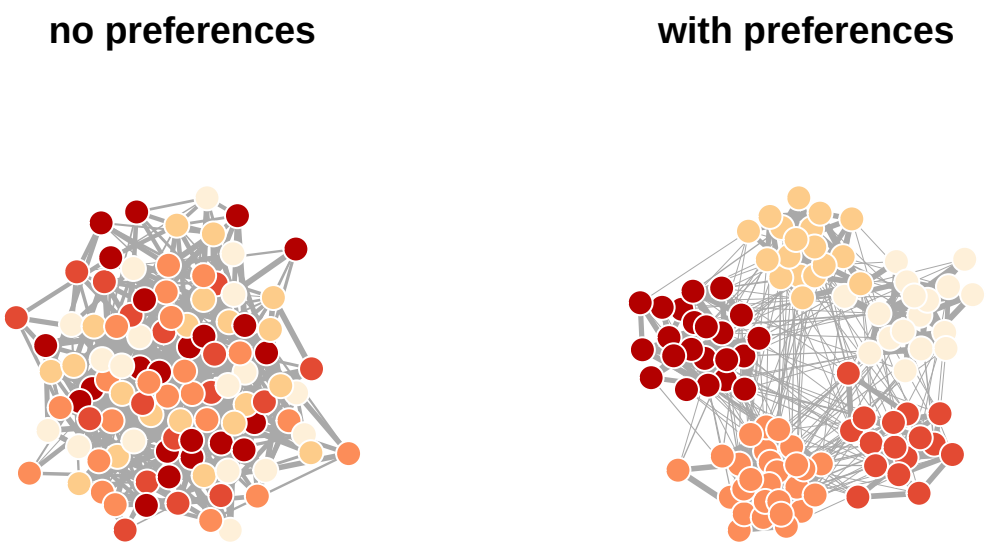
```
preftypes <- 'sim'                                  # preference type. We use similarity preferences.
linktype <- 'stow'                                  # network link type. We use weighted links.
wvalspref <- 0.8                                    # preference weight for the networks with preferences
wvalsnopref <- rep(0, length(wvalspref))            # preference weight for the networks without preferen
onecomp <- TRUE                                     # whether the networks must consist of a single compo
metrics <- c('clust', 'path', 'clustw', 'pathw')    # the network metrics that we want to measure.
nrepls <- 100                                       # the number of networks to be generated for each net
```

Plotting of example networks

Then we plot an example network of each type (i.e. one with and one without the preferences).

To do this, we use the 'traitnet' function of the okaapibeta package, which can generate and plot networks based on different social preferences for traits:

```
par(mfrow = c(1,2))

# a network without preferences
noprefnetres <- traitnet(n, k, traittypes, allncats, preftypes, linktype, wvals = wvalsnopref, onecomp,
title(main = 'no preferences')

# a network with the set preferences
prefnetres <-traitnet(n, k, traittypes, allncats, preftypes, linktype, wvals = wvalspref, onecomp, visn
title(main = 'with preferences')
```



Quantification of network metrics

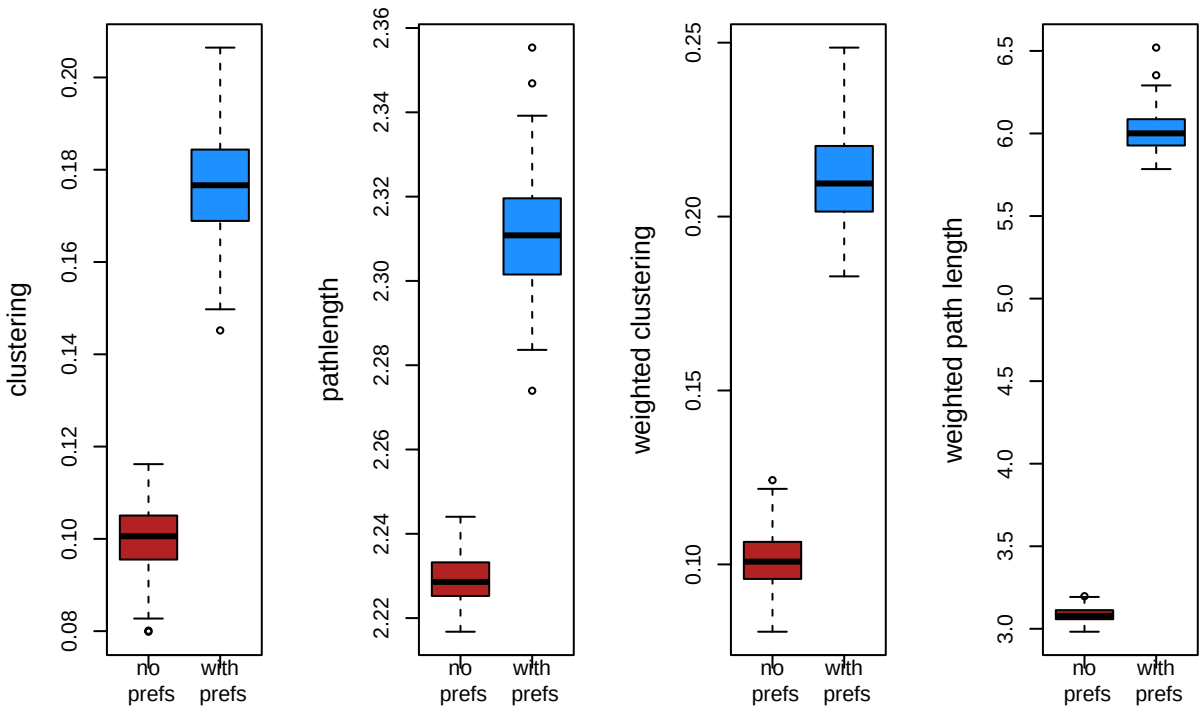
Then we look at how the preferences affect network metrics.

First we measure metrics on network ensembles with and without the preferences. To do this, we use the 'traitnetsmetrics' function of the okaapibeta package, which can generate ensembles of networks based on different social preferences for traits and measure metrics on them:

```
# metrics from networks without preferences
noprefnetmetrics <- traitnetsmetrics(n, k, traittypes, allncats, preftypes, linktype, wvalsnopref, onec

# metrics from networks with the set preferences
prefnetmetrics <- traitnetsmetrics(n, k, traittypes, allncats, preftypes, linktype, wvalspref, onecomp,
```

Plotting of network metrics (results)

Then we plot the metrics, to see whether and how the preferences affect them:



EXAMPLE II: Dynamics on and of animal social networks

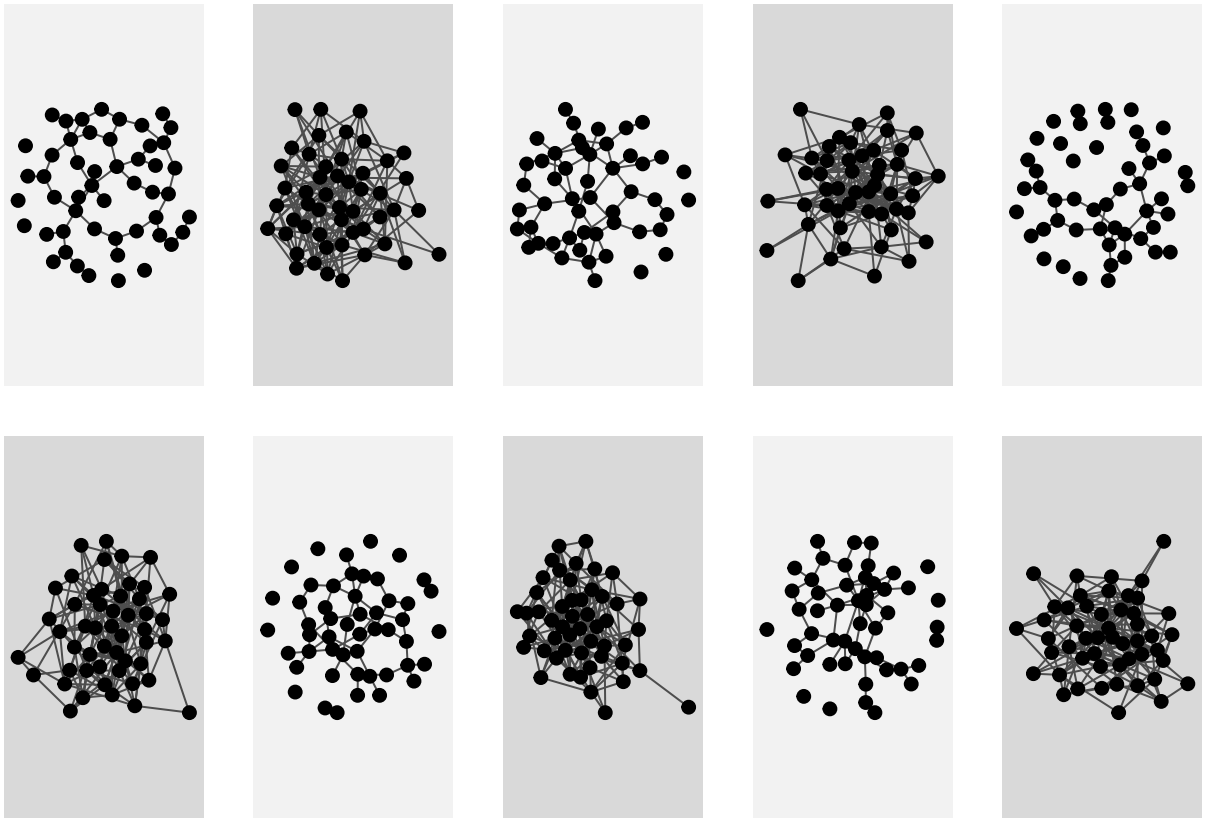
Here, we demonstrate how to use generative models to investigate the link between social network structure and disease spread. First, let's set up a function to generate networks using the Erdös-Rényi algorithm, which produces random networks with a specified link probability:

```r
#Write an algorithm for seasonal network dynamics based on an Erdös-Rényi random graph as an R function
dyn_net_gen<-function(n,p.es,length,seasons,nets_per_season){

  #Creates an empty array to store our dynamic network
  nets<-array(0,dim=c(n,n,length*seasons*nets_per_season))
  #Records the season associated with each network
  seasons2<-rep(rep(seq(1,seasons,1),each=nets_per_season),length)

  #For each network
  for(i in 1:dim(nets)[3]){
    #Run the Erdös-Rényi random graph generator with an edge probability that depends on the season
    net<-igraph::erdos.renyi.game(n=n,p.or.m=p.es[seasons2[i]],type="gnp",directed=FALSE,loops=FALSE)
    #Store the network in our array
    nets[,,i]<-igraph::as_adjacency_matrix(net,type="both",sparse=FALSE)
  }

  #Return our dynamic network object
  return(nets)

}
```

This function has four arguments: `n`, the number of individuals in the network; `p.es`, a vector of season-specific linking probabilities, `length`, the number of "years" to simulate, `seasons`, the number of seasons, and `nets_per_season`, the number of networks to simulate in each season.

We can then use this function to generate a set of networks and plot them to visualize the differences in density.

```r
networks<-dyn_net_gen(n=50,p.es=c(0.05,0.15),length=5,seasons=2,nets_per_season=1)

par(mfrow=c(2,5),mar=c(1,1,1,1))
col_background<-c("gray85","gray95")
for (i in 1:10){
  net<-igraph::graph_from_adjacency_matrix(networks[,,i],mode="undirected",weighted=NULL,diag=FALSE)
  plot(NULL,xlim=c(-2,2),ylim=c(-2,2),xaxt="n",yaxt="n",bty="n",xlab="",ylab="")
  polygon(x=c(-2.5,2.5,2.5,-2.5),y=c(-2.5,-2.5,2.5,2.5),col=col_background[i%%2+1],border=NA)
  par(new=TRUE)
  plot(net,vertex.label=NA,vertex.color="black",edge.color="gray30")
  par(new=FALSE)
}
```



```
par(mfrow=c(1,1),mar=c(5,5,2,2))
```

Now that we have our network generation model, the next thing we need is our model of disease transmission. We'll use a classic discrete time S-I-R model for this purpose. Let's construct some functions to make this easier.

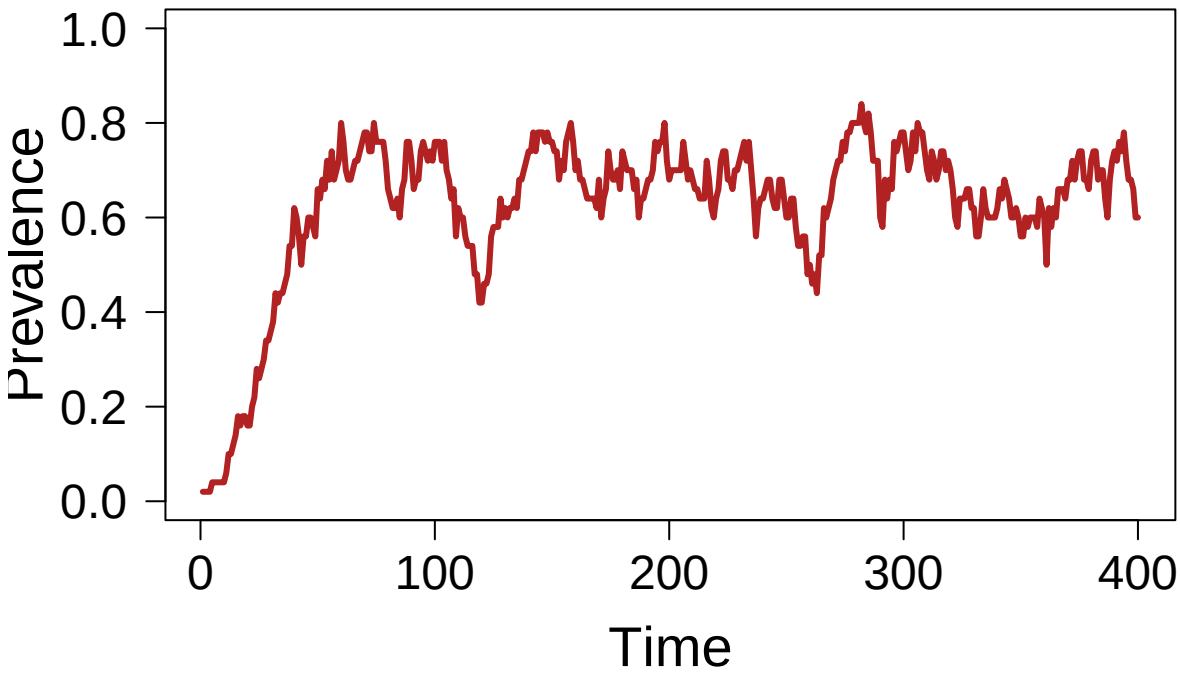
```
#We first need a function that generates information about disease state
pop.gen<-function(n=50,n.I=1){

  pop<-n

  #creates individuals
  indiv.ID<-seq(1,pop,1)

  #creates dataframe containing population info
  indiv.info<-data.frame(indiv.ID)
  names(indiv.info)<-c("ID")

  #create initially infected individuals and generate susceptible (S), infected (I) and recovered (R) i
  I.I<-sample(1:pop,n.I)
  I<-matrix(0,nr=pop,nc=1)
  I[I.I]<-1
  S<-1-I
  R<-matrix(0,nr=pop,nc=1)
```

```
  #combine into dataframe and return dataframe as a list
  indiv.info<-data.frame(indiv.info,S,I,R)
  p<-list(indiv.info)
  return(p)

}

#We then need a function that simulates disease spread
transmit<-function(S,I,mat,si){
  #Pick out rows of adjacency matrix for infected individuals
  pos_inf<-mat[I>0,]
  #Need this adjustment for the case where there is only one infected individual
  #(it just turns the vector produced above back into a matrix)
  if(is.vector(pos_inf)){
    pos_inf<-matrix(pos_inf,nr=1,nc=length(pos_inf))
  }
  #We calculate the infection probabilities for all individuals that an infected individual is connecte
  prob_inf<-pos_inf*si
  #Use binomial draw to work out who gets infected by who
  inf<-array(rbinom(n=length(prob_inf),size=1,prob=prob_inf),dim=dim(prob_inf))
  #This is a convenient way to summarise this per individuals (sign gets rid of multiple infections)
  inf2<-sign(colSums(inf))

  #Assign length of infection to individuals that become infected (the di + 1 is to fit with the functi
  I[inf2==1&S==1]<-1
  #Set the susceptible class value to 0 for infected individuals
  S[inf2==1&S==1]<-0

  #Return new infected and susceptible statuses of all individuals
  output<-list(S,I)
  return(output)
}

#We then need a function that takes the transmit function and simulates the full disease dynamics at ea
timestep<-function(S,I,R,mat,si,ir,rs){

  #Keep track of current infected individuals before transmission modeled
  I_old<-I

  #Model susceptible --> infected [using transmit() function]
  #The if means this bit runs only when there are infected individuals in the population
  if(sum(sign(I))>0){
    #Run the transmission function defined above
    update<-transmit(S=S,I=I,mat=mat,si=si)
    #update new susceptible and infected statuses
    S<-update[[1]]
    I<-update[[2]]
  }

  #Model infected --> recovered
  p_rec<-I_old*ir
  R_new<-rbinom(n=length(R),size=1,p=p_rec)
  R<-sign(R+R_new)
```

```
  I[R_new==1]<-0

  #Model recovered --> susceptible
  p_sus<-R*rs
  S_new<-rbinom(n=length(S),size=1,p=p_sus)
  S<-sign(S+S_new)
  R[S_new==1]<-0

  #Output new classes
  output<-list(S,I,R)
}
```

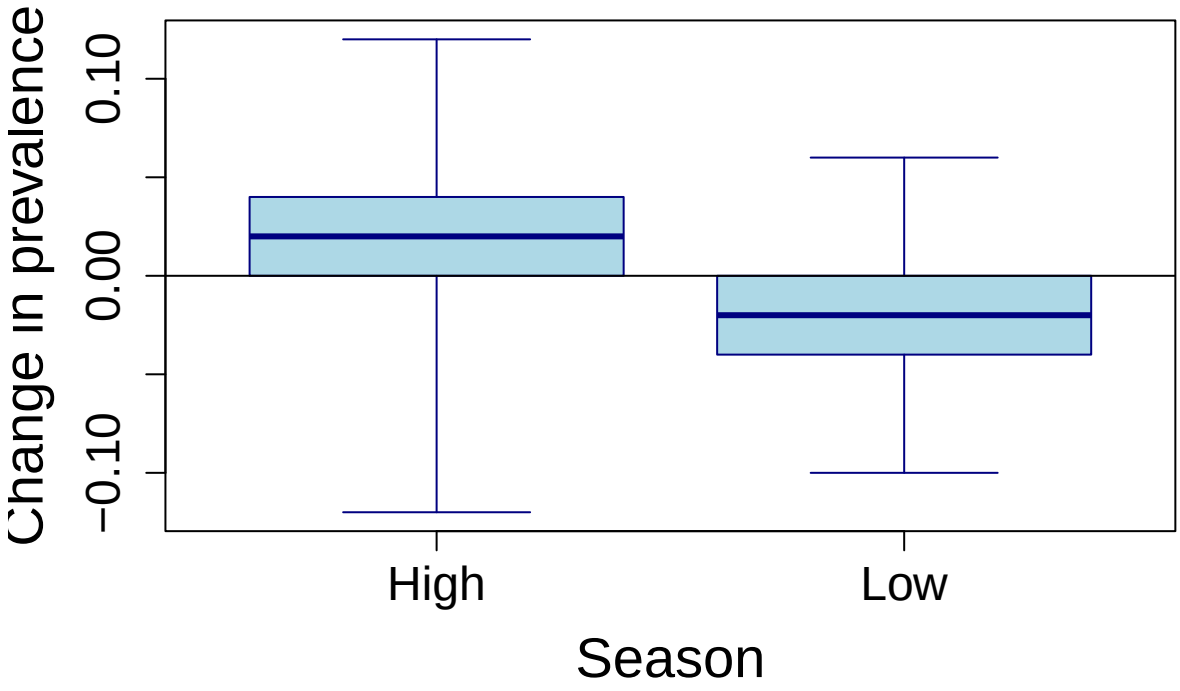
We can then combine our network generation and disease model.

```
#Generate population data
pop<-pop.gen()[[1]]

#Generate seasonally dynamic networks
networks<-dyn_net_gen(n=nrow(pop),p.es=c(0.05,0.15),length=200,seasons=2,nets_per_season=1)

#Set epidemiological parameters
si<-0.03
ir<-0.05
rs<-1

#Vector to store population prevalence over time
prev_summary<-numeric()

#Simulate disease dynamics over the full length of the time period
for(i in 1:dim(networks)[3]){
  #Model disease dynamics
  out<-timestep(S=pop$S,I=pop$I,R=pop$R,mat=networks[,,i],si=si,ir=ir,rs=rs)
  pop$S=out[[1]]
  pop$I=out[[2]]
  pop$R=out[[3]]
  #Store prevalence data
  prev_summary[i]<-sum(pop$I)/nrow(pop)
}

#Plot prevalence over time
plot(prev_summary,type="l",lwd=3,col="firebrick",ylim=c(0,1),las=1,cex.axis=1.5,ylab="Prevalence",xlab=
```



We might be interested in how the network dynamics influence disease dynamics. To test this we could see how changes in prevalence differ between seasons.

```
#Calculate change in prevalence
change_prev<-diff(prev_summary)
#Store information on low connectivity and high connectivity seasons
season<-rep(c("Low","High"),200)
#Remove the first value as we don't have the change for that season
season<-season[-1]

##Plot the change in prevalence as a function of season
##We can see prevalence tends to increase after high connectivity seasons
##and decrease during low connectivity seasons
par(xpd=FALSE)
boxplot(change_prev~season,col="lightblue",border="navy",cex.lab=1.75,cex.axis=1.5,xlab="Season",ylab="
lines(x=c(-100,100),y=c(0,0))
```



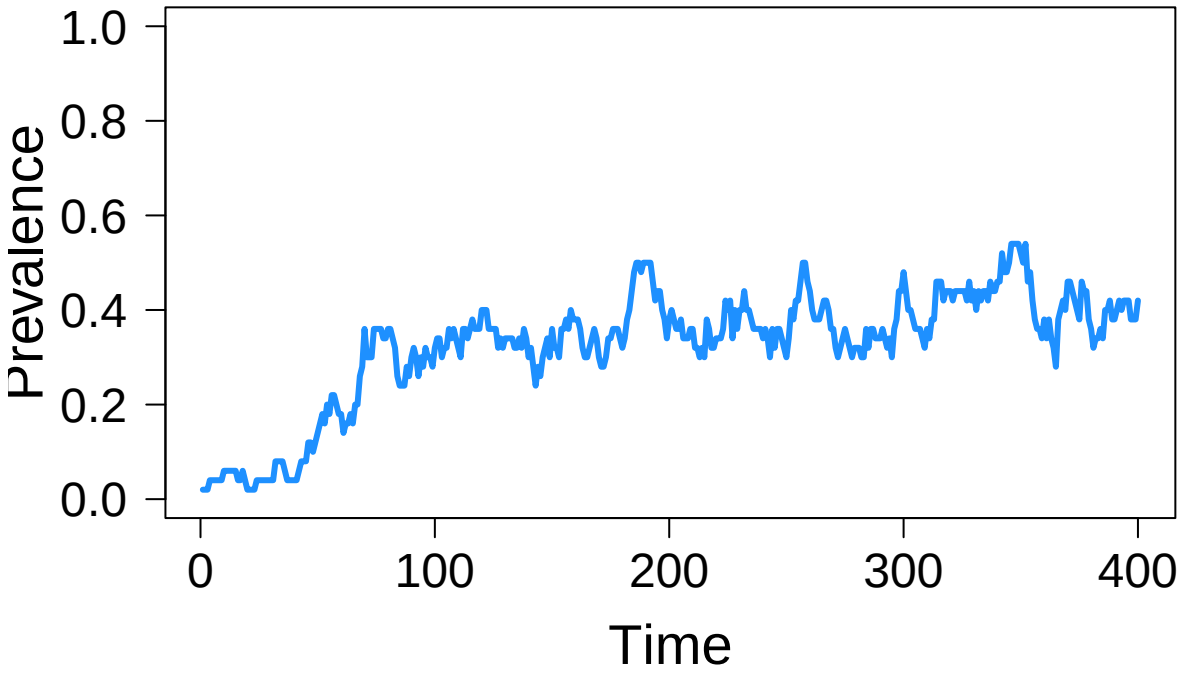
More complicated generative models of network dynamics can introduce dependencies on features of the system. A common example of this in network science is models of individuals changing their social connectivity in response to the spread of infectious disease.

Here we add a new aspect to our generative network model that occurs at each time step. Individuals cut a set proportion of their social connections (defined by p_cut) when prevalence exceeds a certain amount (defined by prev).

```
prev_adj<-function(network,pop,prev,p_cut){
  #Work out who is infected
  t_inf<-which(pop$I==1)
  #if prevalence exceeds threshold
  if(length(t_inf)/nrow(pop)>prev){
    #Find social connections (complete dyads in network)
    conns<-which(network==1,arr.ind=T)
    #(and only keep the undirected information)
    conns<-conns[conns[,1]>conns[,2],]
    #Loop over complete connections
    for(cs in 1:nrow(conns)){
      #Decide if connection will be cut
      flip<-rbinom(1,1,p_cut)
      if(flip==1){
        #Reduce connection strength to negligible amount (i.e. very low chance of transmission)
        network[conns[cs,1],conns[cs,2]]<-network[conns[cs,2],conns[cs,1]]<-0.001
      }
    }
  }
```

```
  return(network)
}
##Note that we don't need to restore connections in our particular situation as we have already
##simulated a new network for each season

#We then need a new version of the time step function that adds the adaptive network dynamics
timestep2<-function(S,I,R,mat,si,ir,rs,pop,prev,p_cut){

  #Keep track of current infected individuals before transmission modeled
  I_old<-I

  #Run network rewiring function
  mat<-prev_adj(network=mat,pop=pop,prev=prev,p_cut=p_cut)

  #Model susceptible --> infected
  #The if means this bit runs only when there are infected individuals in the population
  if(sum(sign(I))>0){
    #Run the transmission function defined above
    update<-transmit(S=S,I=I,mat=mat,si=si)
    #update new susceptible and infected statuses
    S<-update[[1]]
    I<-update[[2]]
  }

  #Model infected --> recovered
  p_rec<-I_old*ir
  R_new<-rbinom(n=length(R),size=1,p=p_rec)
  R<-sign(R+R_new)
  I[R_new==1]<-0

  #Model recovered --> susceptible
  p_sus<-R*rs
  S_new<-rbinom(n=length(S),size=1,p=p_sus)
  S<-sign(S+S_new)
  R[S_new==1]<-0

  #Output new classes
  output<-list(S,I,R)
}

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

##We can now run an example with our new generative model of network dynamics

#Generate population data
pop<-pop.gen()[[1]]

#Generate seasonally dynamic networks
networks<-dyn_net_gen(n=nrow(pop),p.es=c(0.05,0.15),length=200,seasons=2,nets_per_season=1)

#Set epidemiological parameters
si<-0.03
ir<-0.05
```

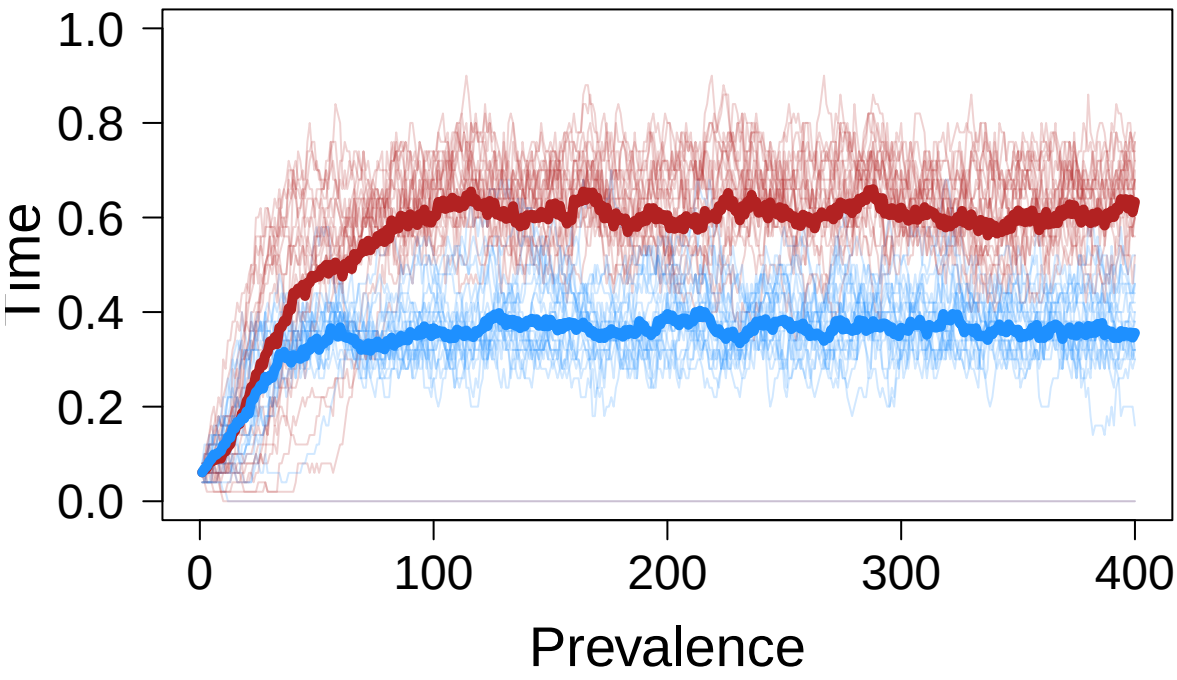
```
rs<-1

#Vector to store population prevalence over time
prev_summary2<-numeric()

#Simulate disease dynamics over the full length of the time period
for(i in 1:dim(networks)[3]){
  out<-timestep2(S=pop$S,I=pop$I,R=pop$R,mat=networks[,,i],si=si,ir=ir,rs=rs,pop=pop,prev=0.3,p_cut=0.5)
  pop$S=out[[1]]
  pop$I=out[[2]]
  pop$R=out[[3]]
  #Store prevalence at each time step
  prev_summary2[i]<-sum(pop$I)/nrow(pop)

}

#Plot prevalence over time
plot(prev_summary2,type="l",lwd=3,col="dodgerblue",ylim=c(0,1),las=1,cex.axis=1.5,ylab="Prevalence",xlab
```



Finally, we can do repeated runs of the model with and without adaptive dynamics to see how this social strategy influences disease transmission.

```
#Set disease transmission parameters
si<-0.03
ir<-0.05
rs<-1
```

```r
#Set up matrices to store prevalence over time in repeat simulations
prev_summary_mat<-prev_summary_mat2<-matrix(0,nr=400,nc=20)

#Simulate 20 times without adaptive network dynamics (just seasonal dynamics)
for(j in 1:20){
  #Set up population
  pop<-pop.gen(n.I=3)[[1]]
  #Set up networks
  networks<-dyn_net_gen(n=nrow(pop),p.es=c(0.05,0.15),length=200,seasons=2,nets_per_season=1)
  #Loop over time steps
  for(i in 1:dim(networks)[3]){
    #Simulate disease dynamics
    out<-timestep(S=pop$S,I=pop$I,R=pop$R,mat=networks[,,i],si=si,ir=ir,rs=rs)
    pop$S=out[[1]]
    pop$I=out[[2]]
    pop$R=out[[3]]
    #Store results
    prev_summary_mat[i,j]<-sum(pop$I)/nrow(pop)
  }
}

for(j in 1:20){
  #Set up population
  pop<-pop.gen(n.I=3)[[1]]
  #Set up networks (without adaptive dynamics)
  networks<-dyn_net_gen(n=nrow(pop),p.es=c(0.05,0.15),length=200,seasons=2,nets_per_season=1)
  for(i in 1:dim(networks)[3]){
    #Simulate disease dynamics and adaptive network dynamics
    out<-timestep2(S=pop$S,I=pop$I,R=pop$R,mat=networks[,,i],si=si,ir=ir,rs=rs,pop=pop,prev=0.3,p_cut=0
    pop$S=out[[1]]
    pop$I=out[[2]]
    pop$R=out[[3]]
    #Store results
    prev_summary_mat2[i,j]<-sum(pop$I)/nrow(pop)
  }
}

##We can then plot the results of the repeat simulation runs to show that prevalence is lower
##when individuals cut their social connections in response to disease being present
plot(NULL,xlim=c(0,400),ylim=c(0,1),xlab="Prevalence",ylab="Time",las=1,cex.axis=1.5,cex.lab=1.75)
for(i in 1:20){
  lines(prev_summary_mat[,i],col=adjustcolor("firebrick",0.2),lwd=1)
  lines(prev_summary_mat2[,i],col=adjustcolor("dodgerblue",0.2),lwd=1)
}
lines(rowMeans(prev_summary_mat),col="firebrick",lwd=5)
lines(rowMeans(prev_summary_mat2),col="dodgerblue",lwd=5)
```



EXAMPLE III: Animal social network robustness

Here, we present an example of how to investigate the robustness of network structures with generative network modelling.

We are interested in the robustness of modular networks (arising from preferences where individuals prefer their own group, such as their own matriline). We use random networks (Erdös-Rényi) and preferential attachment networks (Barabási-Albert) for comparison. To generate the modular networks, we use the trait preference model in the okaapibeta package (see the Set-up section above for information on this package). We use the igraph package to generate the other types of networks.

We measure network robustness by sequentially removing nodes (individuals) from the network until two are left, and recording the time at which the network breaks into more than one component.

We investigate robustness to different types of loss of individuals (node removal), where individuals are removed either in random order or in order depending on their connectedness.

Note: the code produces results for all network types and one type of node removal. To produce results for multiple node removal types (such as in the main text), we would need to run the code one time for each node removal type.

Network generation parameters

First we set parameters which we will use in the network generation.

We set these general parameters:

```
nettypes <- c('ER', 'BA', 'TP')      # network types.'ER'= Erdös-Rényi, 'BA'= Barabasi-Albert, 'TP' = tr
n <- 100                             # number of nodes (individuals) in each network
kapp <- 10                           # approximate average degree (exact for ER and TP networks; necessa
nrepls <- 100                        # number of replicates (networks) of each network type
removaltype <- 1                     # order in which nodes are removed from the network.  1 = random, 2
```

We also need to set some parameters specifically for the trait preference networks. To generate modular networks, we use similarity preferences for a categorical trait:

```
traittypes <- 'cate'     # trait type (we use a categorical trait)
allncats <- 5            # trait category number (we use five trait categories)
preftypes <- 'sim'       # preference type (we use similarity preferences)
wvals <- 0.95            # preference weight
linktype <- 'unw'        # network link type (we use unweighted links)
```

Functions for robustness measurement and plotting

We then write a function to do the sequential removal of nodes to measure robustness (for different types of node removal), as well as a function to plot the results:

```
##### function to measure the robustness of networks to different types of node removal:

netrobustness <- function(net, removaltype){

  ### prepare
  N <- nrow(net) # number of nodes in the network
  removenum <- N-2 # number of nodes to be removed (minus two because components cannot be calculated f
  tmax <- removenum # total number of timesteps
  compnumtzero <- sna::components(net)  # number of network components at time zero (note: igraph also
  compnums <- rep(NA, tmax) # storage for the number of components at each timestep
  curnet <- net # current network
  nodesleft <- seq(N) # names of nodes that are left

  ### find the order of nodes to remove

  if(removaltype == 1){ # nodes are removed in random order
    nodestodie <- sample.int(n = N, size = removenum, replace = FALSE) # names of nodes to be removed,
  }
  else if(removaltype == 2){ # nodes are removed according to degree, with highest degree first
    degs <- rowSums(net) # degrees of all nodes
    nodenamesordered <- order(degs,decreasing = TRUE) # node names ordered according to degree with hig
    nodestodie <- nodenamesordered[1:removenum] # names of nodes to be removed, in order of removal
  }
  else if(removaltype == 3){ # nodes are removed according to degree, with lowest degree first
    degs <- rowSums(net) # degrees of all nodes
    nodenamesordered <- order(degs) # node names ordered according to degree, with lowest degree first
    nodestodie <- nodenamesordered[1:removenum] # names of nodes to be removed, in order of removal
  }

  ### sequentially remove nodes and measure the number of network components

  for (timestep in seq(tmax)) {
```

```
    deadnode <- nodestodie[timestep]                  # name of node to remove
    deadnodeplace <- which(nodesleft==deadnode)       # current row and column of the node to die in the
    curnet <- curnet[-deadnodeplace, -deadnodeplace] # remove the node from the network
    nodesleft <- nodesleft[-deadnodeplace]            # remove the node from the nodesleft vector
    curcompnum <- sna::components(curnet)             # number of network components after the node remo
    compnums[timestep] <- curcompnum                  # store the number of components
  }

  ### prepare and return output

  allcompnums <- c(compnumtzero, compnums) # make output (add the initial component number to the vecto
  return(allcompnums) # return the number of components for each time step (each removal)

} #### end of netrobustness function



##### function that changes the output of the robustness function to a format needed for survival curve

netsurvival <- function(netbreaktimes, n, nrepls) {
  timestepsn <- n-1 # because time 0 is also a timestep
  breaktimestable <- table(netbreaktimes) # a table that gives the number of networks that broke for ti
  stepnames <- as.character(1:timestepsn) # characternames of time steps
  breaknperstepraw <- as.vector(breaktimestable[stepnames]) # breaks per time step where 0 breaks are g
  breaknperstepraw[is.na(breaknperstepraw)] <- 0
  breaknperstep <- breaknperstepraw # breaks per steps where 0 breaks are given by 0
  breakncumsum <- cumsum(breaknperstep) # cumulative sum of breaks across time steps
  survivednets <- nrepls-breakncumsum # networks that are not broken for each time step
  return(survivednets)
}
```

Storage for results

Then we make a matrix to hold the robustness results, with columns representing the different types of network and rows representing replicates.

```
breaktimes <- matrix(nrow = nrepls, ncol = length(nettypes))
colnames(breaktimes) <- nettypes
```

Measuring robustness

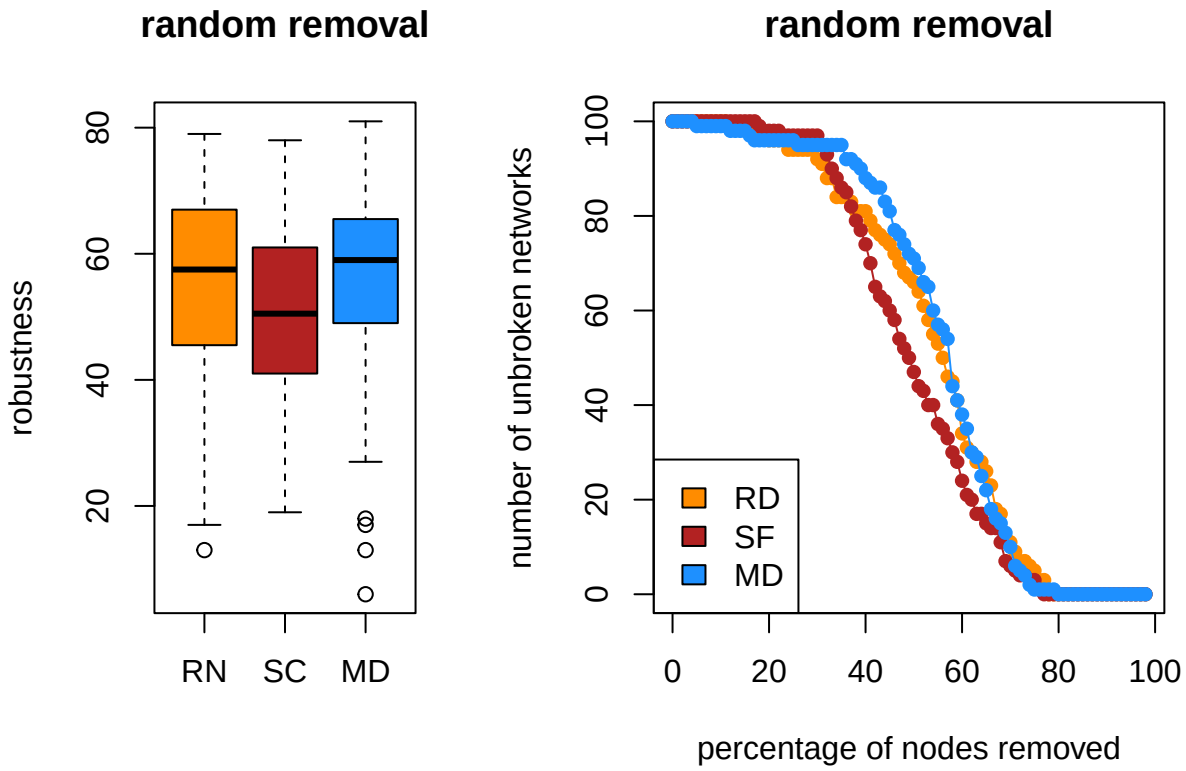
Now, we loop through the different network types and measure their robustness (for many replicates of each type)

```
for(curnettype in nettypes){

  if(curnettype == 'ER'){
    mER <- n*kapp/2 # number of edges that the networks should have

    for(curreplER in 1:nrepls) {
```

```
      # make an ER network
      curnetiER <- sample_gnm(n = n,m = mER) # create an igraph G(n,m) ER network
      curnetER <- as_adjacency_matrix(curnetiER, type = 'both', sparse = FALSE) # change to matrix form

      ## measure and store robustness of the network
      compsovertimeER <- netrobustness(curnetER, removaltype) # the number of components at each time s
      breaktimes[curreplER, curnettype] <- which(compsovertimeER!=1)[1]  # the time the single original

    } # end of ER replicates

  } else if (curnettype == 'BA'){
    mBA <- kapp/2 # number of edges to add in each network generation step

    for(curreplBA in 1:nrepls) {

      # make a BA network
      curnetiBA <- sample_pa(n = n,power = 1, m = mBA, directed = FALSE, algorithm = c("psumtree")) # c
      curnetBA <- as_adjacency_matrix(curnetiBA, type = 'both', sparse = FALSE) # change to matrix form

      ## measure and store robustness
      compsovertimeBA <- netrobustness(curnetBA, removaltype) # the number of components at each time s
      breaktimes[curreplBA, curnettype] <- which(compsovertimeBA!=1)[1] # the time the single original

    } # end of BA replicates

  } else if (curnettype == 'TP'){

    for(curreplTP in 1:nrepls) {

      # make TP network
      netoutputTP <- traitnet(n, kapp, traittypes, allncats, preftypes, linktype, wvals, onecomp = TRUE
      curnetTP <- netoutputTP$net # get the network matrix

      ## measure and store robustness
      compsovertimeTP <- netrobustness(curnetTP, removaltype) # the number of components at each time s
      breaktimes[curreplTP, curnettype] <- which(compsovertimeTP!=1)[1]  # the time the single original

    } # end of TP replicates
  } # end of TP nettype
} # end of all nettypes
```

Plotting robustness results

Finally, we plot the results



EXAMPLE IV: Animal social network methodology

In this final simulation, we'll show how generative network models can be used to assess statistical power in animal social network analysis. Specifically, we'll be investigating how likely a statistical analysis is to detect assortment in the network, given different true levels of assortment and sampling effort.

First, we need to set some parameters for the simulations, using the okaapibeta package (see the Set-up section above for information on the package).

```
n <- 20 # number of individuals
k <- 5 # average degree
traittypes <- "own" # set trait type to "own"
owntraitclasses <- "cate" # the trait is categorical
allncats <- 2 # number of categories
preftypes <- "sim" #preference type is similarity
linktype <- "stow" #set weighted links
onecomp <- TRUE # make networks a single component
visnet <- FALSE # don't visualize the networks
owntraitvals <- rep(c(1,2),n/2) # make trait values
```

The simulation will work by iterating through different combinations of trait preference intensity and sampling effort, so we need to specify these values first. We'll run each combination for 2000 iterations.

```
wvals <- c(0.05,0.1,0.2) #w values
nsamp <- c(5, 10, 20, 50) #number of samples
```

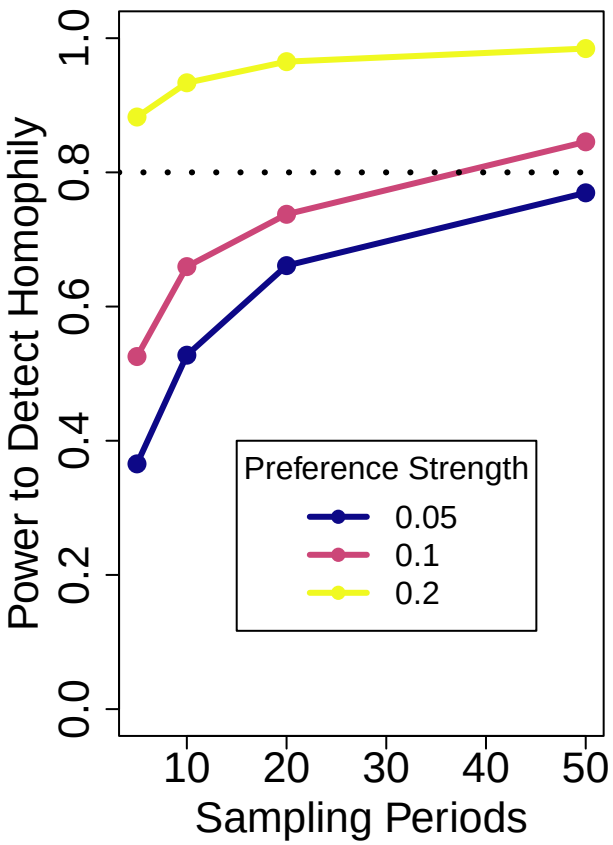
```r
nsim <- 2000 #number of simulations per conditions

res <- array(dim = c(nsim, length(nsamp), length(wvals))) # array to hold results
```

All that's left to do is set up the loop to run the simulation. For each iteration, we'll generate the true network and trait values and simulate binary samples (simulating either scan samples of interactions or association data) from that network. Then, we will use a binomial regression model to test the relationship between having the same trait and the probability of social interactions per sampling period.

```r
for(i in 1:length(wvals)){

  for(j in 1:length(nsamp)){

    res[,j,i] <- replicate(nsim,expr = {
      x <- traitnet(n = n, k = k, traittypes = traittypes, allncats = allncats, preftypes = preftypes, l
      net <- x$net # extract true network
      trait <- x$traitvals[,1] # extract trait values
      same_trait <- sapply(trait, function(z) ifelse(trait == z, 1, 0)) # get a trait similarity matrix
      samp_net <- apply(rgraph(n = n, m = nsamp[j], tprob = net, mode = "graph"),c(2,3),sum) # get samp
      Y <- samp_net[lower.tri(samp_net)] # number of associations/interactions is the response
      D <- nsamp[j] # number of samples serves as denominator
      X <- same_trait[lower.tri(same_trait)] # same trait (0 or 1) is the predictor
      mod <- glm(cbind(Y,D-Y) ~ X, family = binomial) # fit the model
      summary(mod)$coefficients["X","Pr(>|z|)"] # get p-value
    })

  }
}
```

Now we can visualize the results by plotting the power (probability of getting a p-value < 0.05) as a function of both true preference strength and sampling intensity.

```r
cols <- viridis::plasma(3)

par(mar = c(4.5,4.5,1,1), cex.lab = 2, cex.axis = 2)
layout(matrix(c(1,1,1,
                2,3,4), ncol = 2))

plot(colMeans(res[,,1] < 0.05) ~ nsamp, type = "o", pch = 16, cex = 2,  col = cols[1], ylim = c(0,1), lw
points(colMeans(res[,,2] < 0.05) ~ nsamp, type = "o", pch = 16, cex = 2, col = cols[2], lwd = 3)
points(colMeans(res[,,3] < 0.05) ~ nsamp, type = "o", pch = 16, cex = 2, col = cols[3], lwd = 3)

abline(h = 0.8, lty = 3, lwd = 3)

legend(x = 15, y = 0.4, cex = 1.5, legend = wvals, col = cols, title = "Preference Strength", lwd = 3, l

net1 <-traitnet(n = n, k = k, traittypes = traittypes, allncats = allncats, preftypes = preftypes, link
net2 <-traitnet(n = n, k = k, traittypes = traittypes, allncats = allncats, preftypes = preftypes, link
net3 <-traitnet(n = n, k = k, traittypes = traittypes, allncats = allncats, preftypes = preftypes, link
```


Power to Detect Homophily
0.0
0.2
0.4
0.6
0.8
1.0
10
20
30
40
50
Sampling Periods
Preference Strength
0.05
0.1
0.2