Interfacing bnlearn with other R packages

Aaron Smith

6/27/2020

There are a lot of specialized packages in R for working with graphical models. Different packages have different advantages and disadvantages. Different packages have different object classes for storing output. To fully leverage graphical models in R, we need to be able to move between packages.

Much of the code presented here is copied or modified from <https://www.bnlearn.com/>

Here are two books that go deeper into the bnlearn package:

<https://www.bnlearn.com/book-useR/>

<https://www.bnlearn.com/book-crc/>

The first book can be downloaded from SpringerLink through UCF’s library. The two books and website overlap a lot; it will be worth your time to work through all three if you are going to use Bayesian networks professionally.

# Interfacing bnlearn with the deal R package

The deal package is one of the oldest R packages for structure and parameter learning; notably, it supports conditional linear Gaussian networks. It implements a single option for learning: hill climbing with a posterior score followed by posterior estimates of the parameters.

## Exporting a network structure to deal

It is possible to export a network structure learned from data to deal; since deal does not have separate classes for network structures and fitted networks (e.g. like bn vs bn.fit objects in bnlearn), it is not possible to export a network structure that does not have an associated data set, such as one created with modelstring(). Furthermore, deal does not support partially directed graphs, so the network structure must be completely directed.

Since the two packages use the same model string representation, that is the easiest way of exporting the network structure. Both packages have functions called modelstring() that take network models and return their string representation; and as.network() in deal does the opposite, much like model2network() in bnlearn.

library(bnlearn)

##   
## Attaching package: 'bnlearn'

## The following object is masked from 'package:stats':  
##   
## sigma

data(alarm)  
hc\_alarm = hc(alarm)  
modelstring\_alarm <- bnlearn::modelstring(hc\_alarm)  
node\_ordering\_alarm <- bnlearn::node.ordering(hc\_alarm)  
alarm <- alarm[,node\_ordering\_alarm]

Note that deal requires the columns of the data set to follow the topological ordering of the nodes in the graph.

#library(deal)  
deal\_alarm <- deal::network(alarm[,node\_ordering\_alarm])  
deal\_alarm <- deal::as.network(modelstring\_alarm, deal\_alarm)  
deal::modelstring(deal\_alarm)

## [1] "[HIST][HRBP][PAP][FIO2][ANES][ERCA][LVF|HIST][PMB|PAP][ERLO|HRBP][PCWP|LVF][HR|HRBP:ERLO][HREK|ERCA:HR][HRSA|ERCA:HR][LVV|LVF:PCWP][CCHL|HR][CVP|LVV][MINV|CCHL][STKV|LVF:LVV][CO|HR:STKV][HYP|LVV:STKV][VALV|MINV][INT|MINV:VALV][PVS|FIO2:VALV][ACO2|CCHL:VALV][PRSS|VALV:INT][SHNT|PMB:INT][VLNG|MINV:VALV:INT][SAO2|PVS:SHNT][ECO2|ACO2:VLNG][KINK|PRSS:VLNG][VTUB|MINV:INT:PRSS][TPR|CCHL:SAO2][DISC|VTUB][BP|CO:TPR][APL|TPR][VMCH|VTUB:DISC][MVS|VMCH]"

modelstring\_alarm

## [1] "[HIST][HRBP][PAP][FIO2][ANES][ERCA][LVF|HIST][PMB|PAP][ERLO|HRBP][PCWP|LVF][HR|HRBP:ERLO][HREK|HR:ERCA][HRSA|HR:ERCA][LVV|PCWP:LVF][CCHL|HR][CVP|LVV][MINV|CCHL][STKV|LVF:LVV][CO|STKV:HR][HYP|LVV:STKV][VALV|MINV][INT|MINV:VALV][PVS|FIO2:VALV][ACO2|CCHL:VALV][PRSS|INT:VALV][SHNT|PMB:INT][VLNG|MINV:INT:VALV][SAO2|SHNT:PVS][ECO2|ACO2:VLNG][KINK|PRSS:VLNG][VTUB|PRSS:MINV:INT][TPR|SAO2:CCHL][DISC|VTUB][BP|TPR:CO][APL|TPR][VMCH|DISC:VTUB][MVS|VMCH]"

## Importing a network structure from deal

The same technique can be used to import a network structure from deal.

bnlearn::model2network(deal::modelstring(deal\_alarm))

##   
## Random/Generated Bayesian network  
##   
## model:  
## [ANES][ERCA][FIO2][HIST][HRBP][PAP][ERLO|HRBP][LVF|HIST][PMB|PAP]  
## [HR|ERLO:HRBP][PCWP|LVF][CCHL|HR][HREK|ERCA:HR][HRSA|ERCA:HR][LVV|LVF:PCWP]  
## [CVP|LVV][MINV|CCHL][STKV|LVF:LVV][CO|HR:STKV][HYP|LVV:STKV][VALV|MINV]  
## [ACO2|CCHL:VALV][INT|MINV:VALV][PVS|FIO2:VALV][PRSS|INT:VALV][SHNT|INT:PMB]  
## [VLNG|INT:MINV:VALV][ECO2|ACO2:VLNG][KINK|PRSS:VLNG][SAO2|PVS:SHNT]  
## [VTUB|INT:MINV:PRSS][DISC|VTUB][TPR|CCHL:SAO2][APL|TPR][BP|CO:TPR]  
## [VMCH|DISC:VTUB][MVS|VMCH]  
## nodes: 37   
## arcs: 53   
## undirected arcs: 0   
## directed arcs: 53   
## average markov blanket size: 3.46   
## average neighbourhood size: 2.86   
## average branching factor: 1.43   
##   
## generation algorithm: Empty

# Interfacing with the pcalg R package

The pcalg package is a versatile R package for structure learning. It implements both score-based algorithms such as the Greedy Equivalent Search (GES) and constraint-based algorithms such as the PC. It is one of the few R packages that can handle discrete data sets as well as cotinuous data sets. Exporting a network structure to pcalg

Exporting bn or bn.fit objects to pcalg is not currently implemented in bnlearn.

## Importing a network structure from pcalg

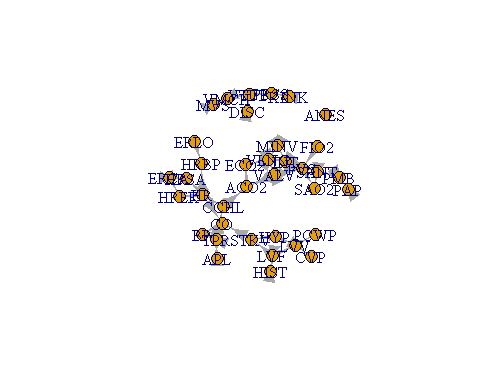
Importing network structures is implementeed in as.bn().

To learn a model with pcalg, you need a conditional independence test.

* pcalg::dsepTest() test for d-separation in a DAG
* pcalg::binCItest() G square Test for conditional Independence of Binary Variable
* pcalg::disCItest() a conditional independence test for discretevariables
* pcalg::gaussCItest() conditional independence test for Gaussian variable

To learn a discrete model with pcalg::pc(), the levels of the variables need to be coded as non-negative integers, 0, 1, 2,….

v\_nlevels <- sapply(X = alarm,FUN = nlevels)  
integer\_alarm <- sapply(  
 X = alarm,  
 FUN = function(x) as.integer(x) - 1  
)  
list\_suffStat <- list(  
 dm = integer\_alarm,  
 nlev = v\_nlevels,  
 adaptDF = FALSE  
)  
pcalg\_alarm <- pcalg::pc(  
 suffStat = list\_suffStat,  
 indepTest = pcalg::disCItest,  
 alpha = 0.01,  
 labels = colnames(alarm)  
)  
pcalg::iplotPC(pcalg\_alarm)



bnlearn::as.bn(pcalg\_alarm)

##   
## Random/Generated Bayesian network  
##   
## model:  
## [partially directed graph]  
## nodes: 37   
## arcs: 41   
## undirected arcs: 9   
## directed arcs: 32   
## average markov blanket size: 2.92   
## average neighbourhood size: 2.22   
## average branching factor: 0.86   
##   
## generation algorithm: Empty

Importing fitted Bayesian networks into bn.fit objects is not supported. Note that as graphs are imported with as.bn() they are checked to be acyclic, unless the user specifies check.cycles = FALSE.

# Interfacing with the graph R package

The graph package is available from Bioconductor and it is one of the most popular packages to work on graphs (both directed and undirected). It implements a variety of algorithms for random graph generation, centrality statistics, graph distances, nodes and arcs manipulation utilities, and it provides a strong foundation for the Rgraphviz package. For this reason, bnlearn implements functions to import and export network structures to graph as:

* graphNEL objects, which encode the graph as a list in which each element refer to one of the nodes in the graph and contains a vector with its children;
* graphAM objects, which encode the graph as a matrix in which the (i, j) cell is equal to one if there is an arc from the ith node to the jth node and zero otherwise.

## Exporting a network structure to graph

Exporting the objects is achieved with the conversion methods as.graphNEL() and as.graphAM().

graphNEL\_alarm <- bnlearn::as.graphNEL(hc\_alarm)  
graphNEL\_alarm

## A graphNEL graph with directed edges  
## Number of Nodes = 37   
## Number of Edges = 53

graphAM\_alarm <- bnlearn::as.graphAM(hc\_alarm)  
graphAM\_alarm

## A graphAM graph with directed edges  
## Number of Nodes = 37   
## Number of Edges = 53

Both methods accept objects of class bn.fit as well as of class bn, and they implicitly call bn.net() on the former to export them.

bn\_alarm <- bnlearn::bn.fit(  
 x = hc\_alarm,  
 data = alarm  
)  
bnlearn::as.graphNEL(bn\_alarm)

## A graphNEL graph with directed edges  
## Number of Nodes = 37   
## Number of Edges = 53

bnlearn::as.graphAM(bn\_alarm)

## A graphAM graph with directed edges  
## Number of Nodes = 37   
## Number of Edges = 53

## Importing a network structure from graph

Importing network structures works in the same way, with a conversion method bnlearn::as.bn().

bnlearn::as.bn(graphNEL\_alarm)

##   
## Random/Generated Bayesian network  
##   
## model:  
## [HIST][HRBP][PAP][FIO2][ANES][ERCA][LVF|HIST][PMB|PAP][ERLO|HRBP][PCWP|LVF]  
## [HR|HRBP:ERLO][HREK|HR:ERCA][HRSA|HR:ERCA][LVV|PCWP:LVF][CCHL|HR][CVP|LVV]  
## [MINV|CCHL][STKV|LVF:LVV][CO|STKV:HR][HYP|LVV:STKV][VALV|MINV][INT|MINV:VALV]  
## [PVS|FIO2:VALV][ACO2|CCHL:VALV][PRSS|INT:VALV][SHNT|PMB:INT]  
## [VLNG|MINV:INT:VALV][SAO2|SHNT:PVS][ECO2|ACO2:VLNG][KINK|PRSS:VLNG]  
## [VTUB|PRSS:MINV:INT][TPR|SAO2:CCHL][DISC|VTUB][BP|TPR:CO][APL|TPR]  
## [VMCH|DISC:VTUB][MVS|VMCH]  
## nodes: 37   
## arcs: 53   
## undirected arcs: 0   
## directed arcs: 53   
## average markov blanket size: 3.46   
## average neighbourhood size: 2.86   
## average branching factor: 1.43   
##   
## generation algorithm: Empty

bnlearn::as.bn(graphAM\_alarm)

##   
## Random/Generated Bayesian network  
##   
## model:  
## [HIST][HRBP][PAP][FIO2][ANES][ERCA][LVF|HIST][PMB|PAP][ERLO|HRBP][PCWP|LVF]  
## [HR|HRBP:ERLO][HREK|HR:ERCA][HRSA|HR:ERCA][LVV|PCWP:LVF][CCHL|HR][CVP|LVV]  
## [MINV|CCHL][STKV|LVF:LVV][CO|STKV:HR][HYP|LVV:STKV][VALV|MINV][INT|MINV:VALV]  
## [PVS|FIO2:VALV][ACO2|CCHL:VALV][PRSS|INT:VALV][SHNT|PMB:INT]  
## [VLNG|MINV:INT:VALV][SAO2|SHNT:PVS][ECO2|ACO2:VLNG][KINK|PRSS:VLNG]  
## [VTUB|PRSS:MINV:INT][TPR|SAO2:CCHL][DISC|VTUB][BP|TPR:CO][APL|TPR]  
## [VMCH|DISC:VTUB][MVS|VMCH]  
## nodes: 37   
## arcs: 53   
## undirected arcs: 0   
## directed arcs: 53   
## average markov blanket size: 3.46   
## average neighbourhood size: 2.86   
## average branching factor: 1.43   
##   
## generation algorithm: Empty

Note that as graphs are imported with as.bn() they are checked to be acyclic, unless the user specifies check.cycles = FALSE.

M <- matrix(0, nrow = 3, ncol = 3)  
M[1, 2] <- 1  
M[2, 3] <- 1  
M[3, 1] <- 1  
cyclic = graph::graphAM(M, edgemode = "directed")  
try(as.bn(cyclic))

## Error in as.bn.graphAM(cyclic) :   
## the graphAM object contains directed cycles.

as.bn(cyclic, check.cycles = FALSE)

##   
## Random/Generated Bayesian network  
##   
## model:  
## [n1|n3][n2|n1][n3|n2]   
## nodes: 3   
## arcs: 3   
## undirected arcs: 0   
## directed arcs: 3   
## average markov blanket size: 2.00   
## average neighbourhood size: 2.00   
## average branching factor: 1.00   
##   
## generation algorithm: Empty

# Interfacing with the igraph R package

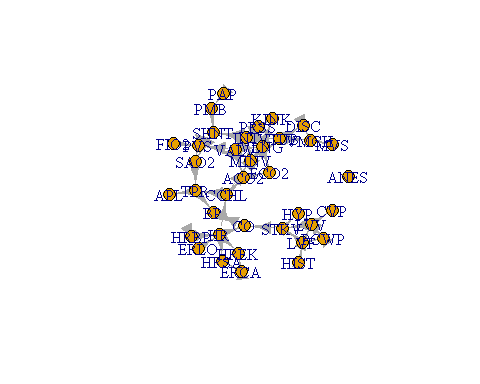
The igraph package is the R interface to the igraph library for network analysis. It implements an extensive selection of algorithms for creating and generating directed and undirected graphs, manipulating nodes and arcs, and it provides highly customizable plotting facilities. For this reason, bnlearn implements functions to import and export network structures to igraph’s native objects, which are themselves of class igraph. Exporting a network structure to igraph

Exporting objects is achieved with the conversion method igraph::as.igraph().

igraph\_alarm <- igraph::graph\_from\_adjacency\_matrix(  
 adjmatrix = bnlearn::amat(hc\_alarm),  
 mode = "directed"   
)  
igraph\_alarm

## IGRAPH 26db28e DN-- 37 53 --   
## + attr: name (v/c)  
## + edges from 26db28e (vertex names):  
## [1] PCWP->LVV HIST->LVF TPR ->BP TPR ->APL CO ->BP HRBP->ERLO  
## [7] HRBP->HR PAP ->PMB SAO2->TPR FIO2->PVS PRSS->KINK PRSS->VTUB  
## [13] MINV->INT MINV->VALV MINV->VLNG MINV->VTUB LVF ->PCWP LVF ->LVV   
## [19] LVF ->STKV PMB ->SHNT INT ->PRSS INT ->SHNT INT ->VLNG INT ->VTUB  
## [25] DISC->VMCH LVV ->CVP LVV ->HYP LVV ->STKV STKV->CO STKV->HYP   
## [31] CCHL->TPR CCHL->MINV CCHL->ACO2 ERLO->HR HR ->CO HR ->HREK  
## [37] HR ->HRSA HR ->CCHL ERCA->HREK ERCA->HRSA SHNT->SAO2 PVS ->SAO2  
## [43] ACO2->ECO2 VALV->PRSS VALV->INT VALV->PVS VALV->ACO2 VALV->VLNG  
## + ... omitted several edges

plot(igraph\_alarm)



Undirected arcs are exported as a pair of directed arcs, that is, Xi — Xj becomes {Xi → Xj, Xi ← Xj}.

igraph::graph\_from\_adjacency\_matrix(  
 adjmatrix = bnlearn::amat(hc\_alarm),  
 mode = "undirected"   
)

## IGRAPH 26eba56 UN-- 37 53 --   
## + attr: name (v/c)  
## + edges from 26eba56 (vertex names):  
## [1] CVP --LVV PCWP--LVF PCWP--LVV HIST--LVF TPR --BP TPR --SAO2  
## [7] TPR --APL TPR --CCHL BP --CO CO --STKV CO --HR HRBP--ERLO  
## [13] HRBP--HR HREK--HR HREK--ERCA HRSA--HR HRSA--ERCA PAP --PMB   
## [19] SAO2--SHNT SAO2--PVS FIO2--PVS PRSS--INT PRSS--KINK PRSS--VALV  
## [25] PRSS--VTUB ECO2--ACO2 ECO2--VLNG MINV--INT MINV--CCHL MINV--VALV  
## [31] MINV--VLNG MINV--VTUB MVS --VMCH HYP --LVV HYP --STKV LVF --LVV   
## [37] LVF --STKV PMB --SHNT INT --SHNT INT --VALV INT --VLNG INT --VTUB  
## [43] KINK--VLNG DISC--VTUB DISC--VMCH LVV --STKV CCHL--HR CCHL--ACO2  
## + ... omitted several edges

## Importing a network structure from igraph

Importing network structures works in the same way, with a conversion method as.bn().

M\_alarm <- igraph::as\_adjacency\_matrix(  
 graph = igraph\_alarm,  
 sparse = FALSE  
)  
bn\_from\_adjacency <- bnlearn::empty.graph(colnames(alarm))  
amat(bn\_from\_adjacency) <- M\_alarm

## Warning in check.amat(amat = value, nodes = names(x$nodes)): rearranging the  
## rows/columns of the adjacency matrix.

bn\_from\_adjacency

##   
## Random/Generated Bayesian network  
##   
## model:  
## [HIST][HRBP][PAP][FIO2][ANES][ERCA][LVF|HIST][PMB|PAP][ERLO|HRBP][PCWP|LVF]  
## [HR|HRBP:ERLO][HREK|ERCA:HR][HRSA|ERCA:HR][LVV|LVF:PCWP][CCHL|HR][CVP|LVV]  
## [MINV|CCHL][STKV|LVF:LVV][CO|HR:STKV][HYP|LVV:STKV][VALV|MINV][INT|MINV:VALV]  
## [PVS|FIO2:VALV][ACO2|CCHL:VALV][PRSS|VALV:INT][SHNT|PMB:INT]  
## [VLNG|MINV:VALV:INT][SAO2|PVS:SHNT][ECO2|ACO2:VLNG][KINK|PRSS:VLNG]  
## [VTUB|MINV:INT:PRSS][TPR|CCHL:SAO2][DISC|VTUB][BP|CO:TPR][APL|TPR]  
## [VMCH|VTUB:DISC][MVS|VMCH]  
## nodes: 37   
## arcs: 53   
## undirected arcs: 0   
## directed arcs: 53   
## average markov blanket size: 3.46   
## average neighbourhood size: 2.86   
## average branching factor: 1.43   
##   
## generation algorithm: Empty

Note that as graphs are imported with as.bn() they are checked to be acyclic, unless the user specifies check.cycles = FALSE.

# Interfacing with the gRain R package

The gRain package is available from CRAN and provides the only implementation of exact inference in R; currently it only supports discrete Bayesian networks. The main data structure in gRain is the grain class, which stores a fitted Bayesian network as a list of conditional probability tables (much like bnlearn’s bn.fit objects) and makes it possible for setEvidence() and querygrain() to perform posterior inference via belief propagation. Exporting a fitted Bayesian networks to gRain

bn.fit objects can be exported with the conversion method as.grain().

grain\_alarm <- bnlearn::as.grain(  
 x = bn\_alarm  
)

## Loading required namespace: gRain

## Warning in as.grain.bn.fit(x = bn\_alarm): NaN conditional probabilities in ECO2,  
## replaced with a uniform distribution.

## Warning in as.grain.bn.fit(x = bn\_alarm): NaN conditional probabilities in VLNG,  
## replaced with a uniform distribution.

## Warning in as.grain.bn.fit(x = bn\_alarm): NaN conditional probabilities in VTUB,  
## replaced with a uniform distribution.

grain\_alarm

## Independence network: Compiled: TRUE Propagated: FALSE   
## Nodes: chr [1:37] "CVP" "PCWP" "HIST" "TPR" "BP" "CO" "HRBP" "HREK" "HRSA" "PAP" ...

Only discrete networks can be exported, since gRain does not support networks with other parametric assumptions.

The other important limitation of gRain, compared to bnlearn, is that it does not allow for conditional probabilities to be NaN. (This happens when estimating them via maximum likelihood and some parent configurations are not observed in the data.) In that case as.grain() will give a warning and replace the NaNs with uniform distributions, much like the Bayesian posterior estimator would.

## Importing a network structure from gRain

Importing network structures works in the same way, with a conversion method as.bn.fit().

grain\_to\_bn\_fit <- as.bn.fit(  
 x = grain\_alarm  
)  
all.equal(bn\_alarm,grain\_to\_bn\_fit)

## [1] "Different probabilities for node ECO2"

For convenience, bnlearn also provides an as.bn() conversion function that returns the network structure underlying the grain object as a bn object.

grain\_to\_bn <- as.bn(  
 x = grain\_alarm  
)  
all.equal(hc\_alarm,grain\_to\_bn)

## [1] TRUE

Note that by default as.bn.fit() disregards any evidence that has been set in the grain object with setEvidence(). So, for instance, setting B equal to b does not have any effect on the conditional probability table for node B in the returned bn.fit object.

setEvidence\_alarm = gRain::setEvidence(  
 object = grain\_alarm,  
 node = "MVS", state = "HIGH"  
)

##   
## Attaching package: 'gRbase'

## The following objects are masked from 'package:bnlearn':  
##   
## ancestors, children, parents

as.bn.fit(setEvidence\_alarm)$MVS

##   
## Parameters of node MVS (multinomial distribution)  
##   
## Conditional probability table:  
##   
## VMCH  
## MVS HIGH LOW NORMAL ZERO  
## HIGH 0.8326180258 0.0042553191 0.0007193382 0.0490797546  
## LOW 0.0085836910 0.8382978723 0.0002997243 0.0419222904  
## NORMAL 0.1587982833 0.1574468085 0.9989809375 0.9089979550