Graphical Simulator Generator

Bob and Associates

2023-05-02

This is a framework to generate simulated data given a causal graph and some evaluation criteria. We use it to implement Judea Pearl’s “Multistage Simpson’s Paradox machine” which is described in Figure 3 of this article on [Understanding Simpson’s Paradox](http://ftp.cs.ucla.edu/pub/stat_ser/r414-reprint.pdf)“.

library(dplyr)

##   
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':  
##   
## filter, lag

## The following objects are masked from 'package:base':  
##   
## intersect, setdiff, setequal, union

library(ggplot2)  
library(bnlearn) # model2network  
library(Rgraphviz)

## Loading required package: graph

## Loading required package: BiocGenerics

##   
## Attaching package: 'BiocGenerics'

## The following object is masked from 'package:bnlearn':  
##   
## score

## The following objects are masked from 'package:dplyr':  
##   
## combine, intersect, setdiff, union

## The following objects are masked from 'package:stats':  
##   
## IQR, mad, sd, var, xtabs

## The following objects are masked from 'package:base':  
##   
## anyDuplicated, append, as.data.frame, basename, cbind, colnames,  
## dirname, do.call, duplicated, eval, evalq, Filter, Find, get, grep,  
## grepl, intersect, is.unsorted, lapply, Map, mapply, match, mget,  
## order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,  
## rbind, Reduce, rownames, sapply, setdiff, sort, table, tapply,  
## union, unique, unsplit, which.max, which.min

##   
## Attaching package: 'graph'

## The following objects are masked from 'package:bnlearn':  
##   
## degree, nodes, nodes<-

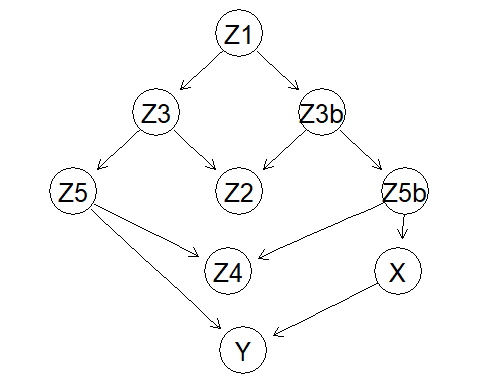
## Loading required package: grid

(The plot of the graph is the only thing that depends on Rgraphviz, which needs to be installed from Bioconductor, or bnlearn which also has some complicated dependencies as I recall. You don’t need these packages to run the simulation.)

# Berkson’s Paradox

Define the causal graph and display it:

msspm\_str <-"[Z1 ]  
 [Z3 | Z1 ]  
 [Z3b| Z1 ]  
 [Z2 | Z3 : Z3b]  
 [Z5 | Z3 ]  
 [Z5b| Z3b ]  
 [Z4 | Z5 : Z5b]  
 [X | Z5b ]  
 [Y | X : Z5 ]" %>% gsub('\\s', '', .)  
  
msspm\_str %>% model2network %>% graphviz.plot



Parse the graph to list the parents of each node.

parse\_graph\_str <- function(graph\_str){  
 parts <- graph\_str %>%  
 strsplit('\\]\\[') %>% '[['(1) %>%   
 gsub('[', '', ., fixed=TRUE) %>%   
 gsub(']', '', ., fixed=TRUE) %>%   
 strsplit('[\\|:] ?')  
   
 setNames(parts %>% sapply('[', -1), nm=parts %>% sapply('[', 1))  
}  
  
node\_parents <- parse\_graph\_str(msspm\_str)  
  
node\_parents

## $Z1  
## character(0)  
##   
## $Z3  
## [1] "Z1"  
##   
## $Z3b  
## [1] "Z1"  
##   
## $Z2  
## [1] "Z3" "Z3b"  
##   
## $Z5  
## [1] "Z3"  
##   
## $Z5b  
## [1] "Z3b"  
##   
## $Z4  
## [1] "Z5" "Z5b"  
##   
## $X  
## [1] "Z5b"  
##   
## $Y  
## [1] "X" "Z5"

Simulate data according to the dependency relationships in the graph, basically as follows: First find the nodes with no parents and give them random values. Then find the nodes whose parents have already been simulated, and give them values using a function that depends on the parents (here we use simple linear functions, but other functions should be possible). Repeat until all nodes have been added as columns in the dataframe of simulated data.

But the graph only tells us which variables depend on which other nodes; it does not tell us their weights. Basically, every edge in the graph can have a weight, and we can pass the set of edge weights into the generation function as parameters. We need to do some gymnastics to get these parameters into a one-dimensional vector, since that is what the optimizer expects. We just put the edges in order alphabetically by the origin and destination.

We also need a loss function to optimize. Since out goal is to make a dataset where including specific sets of covariates gives particular coefficiennts for X, we write a loss function to measure how close we come to achieving that goal.

deserialize\_betas <- function(par, node\_parents){  
 if ( length(par) != length(unlist(node\_parents)))  
 print("Error: length of parameter list does not match length of node parents data structrure.")  
   
 i <- 1  
 beta\_lookup = list()  
 for (node\_name in sort(names(node\_parents))){  
 ## print(sprintf("node name: %s", node\_name))  
 parents <- node\_parents[[node\_name]]  
   
 if (length(parents) > 0){ # skip nodes without parents  
 for (parent\_name in sort(parents)){  
 ## print(sprintf(" parent name: %s", parent\_name))  
 beta\_lookup[[node\_name]][parent\_name] <- par[i]  
 i <- i + 1  
 }  
 }  
  
 }  
   
 beta\_lookup  
}  
  
  
simulation\_function <- function(data\_cols, beta\_vec, noise=0.01){  
 M <- data\_cols %>% as.data.frame %>% as.matrix  
 noise <- rnorm(nrow(M), sd=noise)  
 M %\*% beta\_vec + noise  
}  
  
  
simulate\_with\_pars <- function(par, node\_parents, num\_rows=1000, my\_noise=0.1){  
  
 betas <- deserialize\_betas(par, node\_parents)  
   
 columns <- list()  
   
 remaining\_nodes <- node\_parents  
   
   
 while (length(remaining\_nodes) > 0){  
 for (node\_name in names(remaining\_nodes)){  
 my\_parents <- remaining\_nodes[[node\_name]]  
 if (length(my\_parents) == 0){  
 # no dependencies  
 ## print(sprintf("Adding root node %s", node\_name))  
 columns[[node\_name]] = runif(num\_rows)  
 remaining\_nodes[[node\_name]] <- NULL  
 } else if ( length(setdiff(my\_parents, names(columns))) == 0 ) {  
 # all dependencies are already in the dataset  
 ## print(sprintf("Adding dependent node %s", node\_name))  
 data\_cols <- columns[unlist(my\_parents)]  
 beta\_vec <- betas[[node\_name]][names(data\_cols)]  
 columns[[node\_name]] = simulation\_function(data\_cols, beta\_vec, my\_noise)  
 remaining\_nodes[node\_name] <- NULL  
 } else {  
 print(sprintf("Starting over for node %s", node\_name))  
 }  
 }  
 }  
   
 as.data.frame(columns)  
}  
  
  
get\_coef\_vector\_for\_dataframe <- function(df){  
 c(  
 'just\_X' = coef( lm(Y ~ X, df) )['X'],  
 'add\_Z1' = coef( lm(Y ~ X + Z1, df) )['X'],  
 'add\_Z2' = coef( lm(Y ~ X + Z1 + Z2, df) )['X'],  
 'add\_Z3' = coef( lm(Y ~ X + Z1 + Z2 + Z3, df) )['X'],  
 'add\_Z4' = coef( lm(Y ~ X + Z1 + Z2 + Z3 + Z4, df) )['X'],  
 'add\_Z5' = coef( lm(Y ~ X + Z1 + Z2 + Z3 + Z4 + Z5, df) )['X']  
 )  
}  
  
  
v\_score <- function(coef\_vector){  
 # This is basically Root Mean Squared Error with an extra penalty for getting the signs wrong.  
 target\_vector <- c(1, -1, 1, -1, 1, -1 ) #? c(1, -2, 3, -4, 5, -6 )  
 sign\_penalty <- sum(10 \* (sign(coef\_vector) != sign(target\_vector)))  
 rmse <- sqrt(mean((coef\_vector - target\_vector)^2))  
 sign\_penalty + rmse  
}  
  
  
score\_parameters <- function(par, ...){  
   
 df <- simulate\_with\_pars(par, node\_parents, ...)  
   
 coef\_vector <- get\_coef\_vector\_for\_dataframe(df)  
   
 v\_score(coef\_vector)  
}

Find the optimal parameters using R’s built-in optimizeer:

set.seed(42)  
  
par <- runif(11, min=-0.1, max=0.1)  
  
results <- optim(par, score\_parameters, method='SANN',   
 control=list(maxit=5000))  
  
results$par %>% simulate\_with\_pars(node\_parents, num\_rows=1000) %>% get\_coef\_vector\_for\_dataframe

## just\_X.X add\_Z1.X add\_Z2.X add\_Z3.X add\_Z4.X add\_Z5.X   
## 1.1343070 -0.9723563 0.9107212 -1.0644114 1.1480119 -1.0065171

This is what the resulting dataframe looks like:

mssp <- simulate\_with\_pars(results$par, node\_parents, num\_rows=1000, my\_noise=0.1)  
write.csv(mssp, "multi\_stage\_simpsons\_paradox\_data.csv")  
  
mssp %>% head

## Z1 Z3 Z3b Z2 Z5 Z5b  
## 1 0.17332525 0.814762426 -0.872907261 0.17950076 3.21262512 -4.96518756  
## 2 0.44766812 1.993241072 -2.069572051 0.37231826 7.57739175 -12.27702436  
## 3 0.01891299 0.006397774 -0.008924944 0.02191325 0.06402705 0.09214518  
## 4 0.82177876 3.639697241 -3.951359920 1.17753312 13.68985280 -23.11176160  
## 5 0.29421291 1.306382193 -1.349845921 0.25691080 4.84911034 -7.80959810  
## 6 0.67765093 2.906958029 -3.323507075 1.42669729 10.96710427 -19.34116883  
## Z4 X Y  
## 1 -27.1599977 7.60094129 10.1488240  
## 2 -68.2850485 18.77877787 22.7349421  
## 3 0.8582519 -0.06357441 0.4940198  
## 4 -130.4223714 35.18342547 40.1518240  
## 5 -43.1149142 11.86936892 14.7519686  
## 6 -110.4659095 29.57980796 30.7192693

Now we can test whether the reversals we are looking for in the coefficient of ‘X’ actually happen. If the confounding is correctly implemented, we expect to see the sign of the coefficient of ‘X’ flip back and forth between negative and positive as we add the covariates to the analysis in order.

coef( lm(Y ~ X, mssp) )['X']

## X   
## 1.137793

coef( lm(Y ~ X + Z1, mssp) )['X']

## X   
## -0.998305

coef( lm(Y ~ X + Z1 + Z2, mssp) )['X']

## X   
## 1.01994

coef( lm(Y ~ X + Z1 + Z2 + Z3, mssp) )['X']

## X   
## -1.034066

coef( lm(Y ~ X + Z1 + Z2 + Z3 + Z4, mssp) )['X']

## X   
## 1.281207

coef( lm(Y ~ X + Z1 + Z2 + Z3 + Z4 + Z5, mssp) )['X']

## X   
## -1.007758