DBayesNet

Contents


```
library(DBayesNet)
```

Introduction

DBayesNet provides some helper functions to allow easy structure learning of dynamic Bayesian networks in R. A dynamic Bayesian network (DBN) is a special case of a Bayesian network with temporal dependencies. Whereas a static Bayesian network (BN) represents dependence relationships between a set of variables, a DBN includes extra sets of lagged nodes from the minimum to maxim time lag (known as a Markov lag), representing the past of each variable. A few simple assumptions are utilised to constrain the DBN. Firstly, influence must be extended forwards in time. Secondly a variable at a must be dependent on itself at all previous lags in the network. Once learned a DBN may be 'rolled' up, to represent influence between nodes at any time lag. Unlike a static BN a rolled DBN can represent cyclic relationships and equivalence classes are not a concern as the direction of influence is always forwards in time.

The package `bnlearn` has extensive functionality for static Bayesian network analysis, but does not have functionality for dynamic networks. This utilises `bnlearn`'s structure learning with some additional constraints to learn Dynamic Bayesian network structures. For most uses the `dbn_learn()` function will suffice, but for more specialised applications other functions called by `dbn_learn()` are available to allow production of your own dynamic structure leaning pipeline.

Setup

`DBayesNet` can be installed using `remotes`:

And loaded:

```
library(DBayesNet)
```

Input data

`dbn_learn()` accepts a dataframe with variables in columns and observations in rows. Observations are assumed to be in temporal order. Here we will use some toy data:

```
head(data)
#>              A           B             C
#> 1 -0.07496747 -0.04814249  0.0432159968
#> 2 -0.12600398 -0.08357544  0.0566717323
#> 3 -0.08141014 -0.05506191 -0.0391140190
#> 4 -0.11433833 -0.13671885 -0.0843135552
#> 5 -0.13669822 -0.07686725 -0.0292111403
#> 6 -0.08062770 -0.06759997  0.0003821886
```

Structure learning

The simplest form of DBN utilises a minimum and maximum Markov lag of one, meaning all learned arcs represent influence exerted a single time-step in the future. By default, `dbn_learn()` will learn this type of structure using a hill-climb search:

```
dnet <- dbn_learn(data,score = 'aic-g')
print(dnet)
#>
#>  Dynamic Bayesian Network (dbn)
#>
#>     Max Markov lag: 1
#>     Min Markov lag: 1
#>     Number of variables: 3
#>     Number of arcs in rolled model: 4
#>     Rolled model:
#>  [C|B][B|A:C][A|C]
```

But more complex DBN structures can also be learned through changing the `maxMarkovLag` and `minMarkovLag` parameters:

```
dbn_learn(data,score = 'aic-g',minMarkovLag = 2,maxMarkovLag = 4)
#>
#>  Dynamic Bayesian Network (dbn)
#>
#>     Max Markov lag: 4
#>     Min Markov lag: 4
#>     Number of variables: 3
#>     Number of arcs in rolled model: 6
#>     Rolled model:
#>  [B|A:C][C|B:A][A|C:B]
```

`bnlearn` also offers a tabu search, which can also be accessed using the search parameter:

```
dbn_learn(data,score = 'aic-g',search = "tabu")
#>
#>  Dynamic Bayesian Network (dbn)
#>
#>     Max Markov lag: 1
#>     Min Markov lag: 1
#>     Number of variables: 3
#>     Number of arcs in rolled model: 4
#>     Rolled model:
#>  [C|B][B|A:C][A|C]
```

Parameters can also be passed to `bnlearn`'s `hc()` or `tabu()`, which are used for structure learning under the-hood dependent on the value of search. For example, to perform a hill-climb search for the dynamic structure with random restarts:

```
dbn_learn(data,score = 'aic-g',search = "hc",restart=10)
#>
#>  Dynamic Bayesian Network (dbn)
#>
#>     Max Markov lag: 1
#>     Min Markov lag: 1
#>     Number of variables: 3
#>     Number of arcs in rolled model: 4
#>     Rolled model:
#>  [C|B][A|C][B|C:A]
```

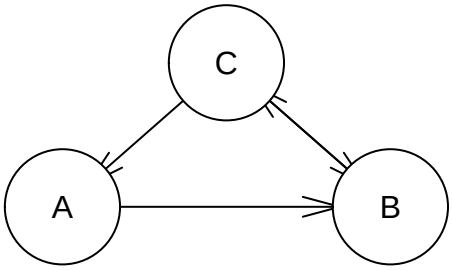
Understanding learned structures

`dbn_learn()` return an object of `dbn` class. `DBayesNet` provides some basic functions to aid in interpretation of the learned DBN structure. As shown in the previous example, calling `print()` on an object of class `dbn` will print some basic information on the rolled up structure of the DBN:

```
print(dnet)
#>
#>  Dynamic Bayesian Network (dbn)
#>
#>     Max Markov lag: 1
#>     Min Markov lag: 1
#>     Number of variables: 3
#>     Number of arcs in rolled model: 4
#>     Rolled model:
#>  [C|B][B|A:C][A|C]
```

We can also plot our DBNs structure. By default this will plot the rolled version of the structure:

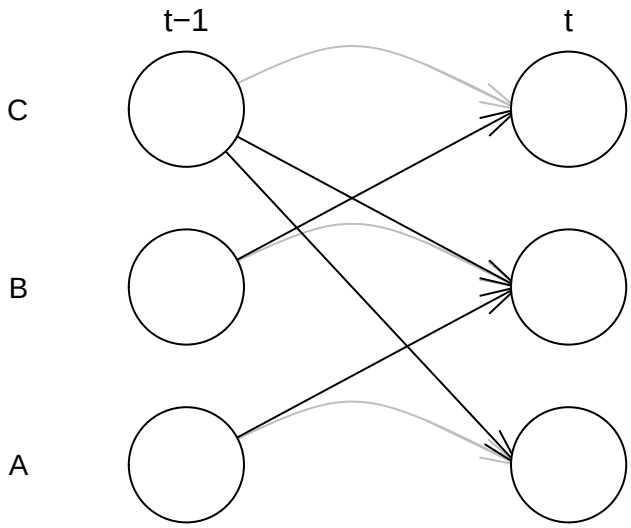
```
plot(dnet)
```



>The arrow argument can be used to control the distance between the arrow head and center of node. This may need to be fine-tuned for different graphs.

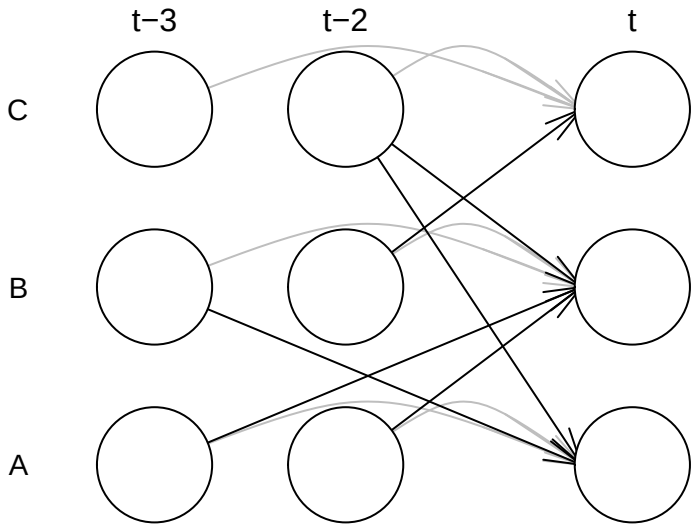
We can also plot the unrolled structure, including whitelisted connections in grey using:

```
plot(dnet,roll=FALSE,whitelist=TRUE)
```



This is particularly useful when plotting structures with more complex Markov lags:

```
dnet2 <- dbn_learn(data,score = 'aic-g',minMarkovLag = 2,maxMarkovLag = 3)

plot(dnet2,roll=FALSE,whitelist=TRUE)
```



An adjacency matrix representing the rolled up structure, with parent nodes in columns and child nodes in rows, can be generated using:

```
as.adjacency.dbn(dnet)
#>   A B C
#> A 1 0 1
#> B 1 1 1
#> C 0 1 1
```

Note that self connections are present in the adjacency matrix by definition of a DBN due to the whitelisted arcs.

Finally, `arc.strength.dbn` is the dynamic equivalent of `bnlearn`'s `arc.strength`.

```
arc.strength.dbn(dnet2,data)
#>    from to lag     strength
#> 1     A  A   2 -42.30972360
#> 2     A  A   3   0.99790304
#> 3     B  B   2 -38.93902607
#> 4     B  B   3   0.58893069
#> 5     C  C   2   0.05014114
#> 6     C  C   3   0.98120377
#> 7     A  B   2 -63.88242427
#> 8     B  C   2 -47.59636511
#> 9     A  B   3  -5.45157211
#> 10    C  B   2  -3.25725586
#> 11    C  A   2  -2.49866016
#> 12    B  A   3  -0.27148789
```

Accessing bn object:
The dbn class contains the bn object in its second element. You can therefore access all bnlearn functionality:

```
print(dnet[2])
#> $bn
#>
#>   Bayesian network learned via Score-based methods
#>
#>   model:
#>    [A_(t-1)][B_(t-1)][C_(t-1)][A_(t)|A_(t-1):C_(t-1)]
#>    [B_(t)|A_(t-1):B_(t-1):C_(t-1)][C_(t)|B_(t-1):C_(t-1)]
#>   nodes:                                 6
#>   arcs:                                  7
#>     undirected arcs:                     0
#>     directed arcs:                       7
#>   average markov blanket size:           3.33
#>   average neighbourhood size:            2.33
#>   average branching factor:              1.17
#>
#>   learning algorithm:                    Hill-Climbing
#>   score:                                 AIC (Gauss.)
#>   penalization coefficient:              1
#>   tests used in the learning procedure:  21
#>   optimized:                             TRUE
#plot(dnet[[2]])
```

Full pipeline

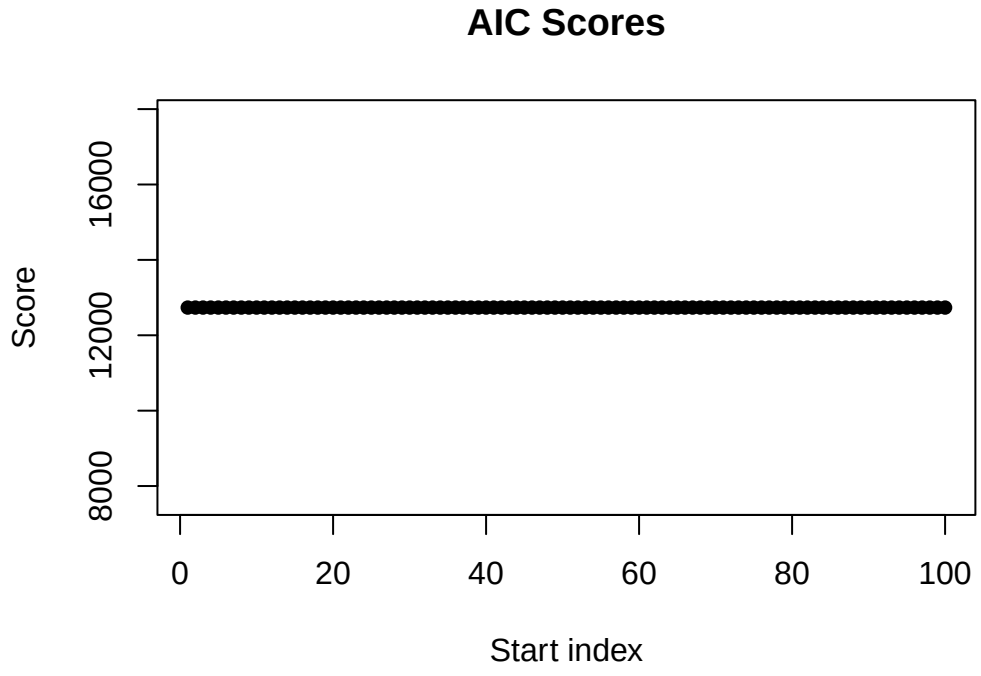
`dbn_learn` wraps a number of functions which facilitate DBN structure learning: 1. `make_dynamic_data()` Coherences data with variables in columns and observations in rows into a dataframe with nVars * (maxMarkovLag - minMarkovLag + 1) columns, where the new columns contain lagged versions of the origional variables from the minimum to maximum Markov Lag. 1. `make_wb_lists()` Produces the whitelist and blacklist to pass to a `bnlearn` structure learning function to force and prevent the necessary arcs for the output to be a dbn. 1. `roll_dbn()` rolls an object of `bn` which represents a dynamic structure into an object of class `bn`.

Column naming in unrolled data follows the convention X_t-L, where X is the variable name and L the lag, other than at lag 0 which uses the convention X_t. It is vital that this convention is maintained throughout the pipeline for the functions to operate properly.

Access to the components in this pipeline allow customised DBN structure learning analysis to be performed. For example, keeping track of the solution space whilst using random restarts:

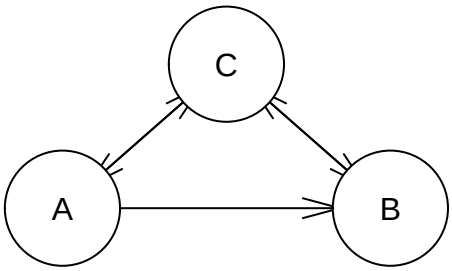
```
library(bnlearn)
#make unrolled dynamic data
dData <- make_dynamic_data(data,minMarkovLag = 1, maxMarkovLag =2)

#make whitelist and blacklist
wb<- make_wb_lists(colnames(data),minMarkovLag =1,maxMarkovLag =2)

whitelist <- wb[[1]]
blacklist <- wb[[2]]
```

```r
#generate random structures for hill-climb starts
start = random.graph(nodes = colnames(dData), num = 100, method = "ordered")

#hill-climb from each random start point, saving results and scores
RES <- lapply(start, function(net) {
  res <- hc(dData,
            whitelist = whitelist,
            blacklist = blacklist,
            score = 'aic-g',
            start = net)
  return(list(score = score(res, dData, type = "aic-g"),
              res = res))
})
scoresList <- lapply(RES, function(x) x$score)
resList <- lapply(RES, function(x) x$res)

#plot scores

plot(unlist(scoresList),
     type = "p",
     pch = 16,
     xlab = "Start index",
     ylab = "Score",
     main = "AIC Scores")
```



We are finding a solution with the same score regardless of where in the search space we start - this suggests that we are accessing the global optimum! A few simple checks (not shown here) will show us that these solutions are all equivalent graphs. We may now want to roll our bn object into a dbn object:

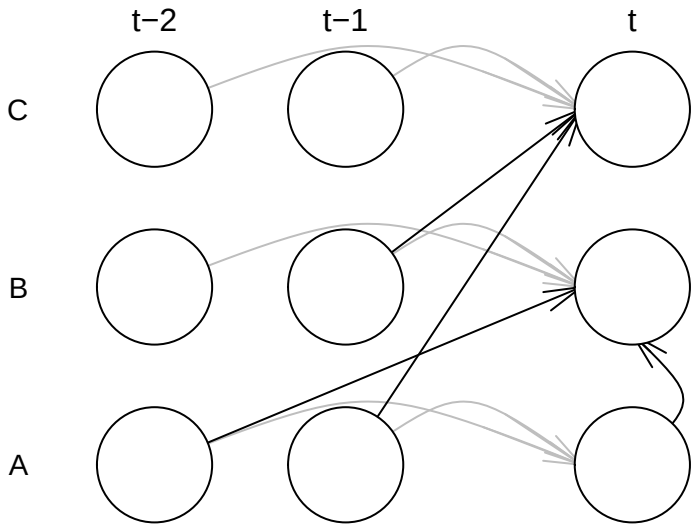
```
dnet3 <- roll_dbn(resList[[1]])

plot(dnet3,roll=T)
```



Intra-slice connections

If the minimum Markov lag is set to 0 in these analysis, the functions will still operate but the output will not be a true DBN. This can be useful when the lag of the system is faster than the sampling resolution, for example in fMRI data. Importantly, the network will not be able to represent cyclic relationships within the same time-slice.

```
dnet4 <- dbn_learn(data2,score='aic-g',minMarkovLag=0,maxMarkovLag = 2)
plot(dnet4,roll=F)
```


t−2
t−1
t
C
B
A