How To Use the Catnet Package

Nikolay Balov, Peter Salzman

March 5, 2010

Introduction

Catnet package implements categorical variable Bayesian network (also called discrete Bayesian network) framework in R. Bayesian networks are graphical statistical models that represent directed dependencies between random variables and thus are able to model causal relationships among these variables. A Bayesian network has two components: Directed Acyclic Graph (DAG) with nodes the variables of interest and a probability structure given as a set of conditional distributions, one for each node in the graph. Any Bayesian network also satisfies the so called *local Markov property*, which states that each node in the network is independent of its non-descendants given its parent nodes. That property leads to a direct factorization of the joint distribution of the node-variables, a fact of great practical importance that is responsible for the popularity of the Bayesian network models.

Two classes of Bayesian networks are among the most used in practice: linear Gaussian networks and categorical ones. In a linear Gaussian network, the nodes represent continuous variables with Gaussian conditional distributions and the expected value of each node is a linear combination of its parent nodes. Gaussian networks benefit from nice analytical properties, but in practice, the linearity and normality assumptions for the observed variables are rarely justified, which limits their applicability. In a categorical Bayesian network, each node takes values in a fixed set of categories and the conditional distributions are multinomial ones with no additional parametric constrains. Consequently, provided that given data is categorical or can be categorized without substantial loss of information, the discrete network framework for modeling the data is a far more general setting - the main reason to be chosen as statistical model in the *catnet*.

The main goal of the package is to provide some tools for inferring discrete Bayesian networks from data based on the Maximum Likelihood criterion. The problem of learning Bayesian networks has relatively long history with abundance of literature devoted to its subject. See for example [9], [7], [6], [12], [15], and some more recent articles, [23], [25], [8]. Although the name Bayesian often implies Bayesian inference, in this package the authors follow a frequentist approach without making assumption on the conditional distributions in form of priors. Since most of the existing network learning algorithms and the corresponding software packages, such as [20], are Bayesian, *catnet* provides tools that are not readily available. Two main techniques are implemented - finding the best network fitting some data for a predefined node order and stochastic search of optimal networks without making assumptions about the node order. For a given node order, an efficient exhaustive search is implemented and the exact MLE solution is given. The approach undertaken is similar to that in [14] without, however, being Bayesian. *Catnet* equips the user not only with graph structure learning abilities but also with probability estimation and prediction utilities. The motivation is to provide more versatile statistical tools for studying networks such as model selection in classes of optimal networks with varying complexity, as proposed in [19]. Despite the diversity and efficiency of the existing algorithms, being strictly structure learning techniques, their selection flexibility is usually very limited. In conclusion, *catnet* does not intend to replace the state of the art learning algorithms but to provide alternative tools for statistical inference.

Although *catnet* is designed as a self-contained package and provides the basic functionality one needs for working with categorical Bayesian networks, some graph related operations such as network visualization are not included. Thus the user may benefit from having installed other packages such as *graph*, for some general graph-related operations, and *Graphviz* or *igraph* for graph rendering and visualization.

The complexity of the problem of learning categorical Bayesian networks is related to the size of the space of such networks, which is super-exponential of the number of nodes. Moreover, it is a NP-Complete problem. Nevertheless, the package makes a step toward easing that problem. When coupled with *snow* package, *catnet* provides parallel processing capabilities allowing the user to benefit from substantial increase in performance.

Although the authors of *catnet* are having in mind reconstruction of gene/protein regulatory networks as a primary application of the package (see for overview [21], [13]), its design is by no means limited for use to bioinformatics only and hopefully will find much larger application scope.

For more detailed introduction to Bayesian networks and different learning strategies, we refer to [16], [17], [9] and [18]. More information on the package functionality and usage can be found in the manual pages.

1 Creating and Manipulating Networks

The basic class object in *catnet* package is `catNetwork`, which stands for categorical network, another name for discrete Bayesian network. Any `catNetwork` can handle different number of categories for its nodes. Its graph structure, a DAG, describes the relationship between its nodes, while multinomial distributions represent the conditional dependency of the nodes on their parents. For brevity, hereafter we will refer to a `catNetwork` object simply as a network.

We start by describing several ways to create a `catNetwork` object. In the usual scenario the *catnet* is designed for, networks are inferred from data and are created implicitly. However, there are occasions, for simulation purposes for example, when the user may want to create a network manually and the package provides such means.

The user can create categorical networks explicitly by calling `cnNew` function. The function takes following arguments: a vector of node names (`nodes`), a list of node categories (`cats`), a list of parents ((`parents`)) and an optional list of conditional probabilities (`probs`). Because of the nested list hierarchy of the probability structure, specifying the probability argument directly can be very elaborated task for large networks. In the following example we create a small network with only three nodes. Note that all inner most vectors in the `probs` argument, such as `(0.4,0.6)`, represent conditional distributions and thus sum to 1. If `probs` parameter is omitted, a random probability structure will be assigned.

```
> library(catnet)
> cnet <- cnNew(nodes = c("a", "b", "c"), cats = list(c("1", "2"),
+     c("1", "2"), c("1", "2")), parents = list(NULL, c(1), c(1,
+     2)), probs = list(c(0.2, 0.8), list(c(0.6, 0.4), c(0.4, 0.6)),
+     list(list(c(0.3, 0.7), c(0.7, 0.3)), list(c(0.9, 0.1), c(0.1,
+         0.9)))))
```

1.1 Generating Random Networks

Randomly generated networks can be useful for simulation and evaluation purposes. By calling `cnRandomCatnet` function, the user may generate a `catNetwork` with random DAG and probability model. The number of nodes, maximum parent size and the number of categories have to be given. All nodes are assigned equal number of categories.

```
> set.seed(123)
> cnet1 <- cnRandomCatnet(numnodes = 4, maxParents = 2, numCategories = 2)
> cnet1

A catNetwork object with  4  nodes,  2  parents,  2  categories,
 Likelihood =  0 , Complexity =  8 .
```

1.2 Inheriting a Graph Object

A `catNetwork` object can be also created by inheriting an existing `graph` object as supported in *graph* package, [10]. The later provides greater number of function for creating and manipulating graphs. A `graph` object can be created directly by specifying its nodes (`myNodes`) and edges (`myEdges`). It contains only a graphical structure description, not a probability one. Then, a `catNetwork` is created by calling `cnCatnetFromGraph` function.

```
> bgraph <- FALSE
> try(bgraph <- require(graph), TRUE)
> if (bgraph) {
+     myNodes <- c("a", "s", "p", "q", "r", "t", "u")
+     myEdges <- list(a = list(edges = NULL), s = list(edges = c("p",
+         "q")), p = list(edges = c("q")), q = list(edges = c("r")),
+         r = list(edges = c("u")), t = list(edges = c("q")), u = list(edges = NULL))
+     g <- new("graphNEL", nodes = myNodes, edgeL = myEdges, edgemode = "directed")
+     cnet2 <- cnCatnetFromGraph(g)
+ }
```

catnet package is able to import Simple Interaction Format (SIF) files. If a SIF file describes a DAG, which may not be the case since SIF files can describe any graph structure, the graph can be imported by calling `cnCatnetFromSif` function.

2 Accessing Network Attributes and Characteristics

There are several functions that give access to the main components of a `catNetwork` object, or its slots to be more correct technically. Such are the functions `cnNumNodes`, `cnNodes`, `cnEdges`, `cnMatEdges`, `cnParents`, `cnMatParents` and `cnProb`. Next we give a short description of them.

Of course, all attributes can be accessed using the `attribute` mechanism, but that opens the possibility of accidental attribute change. Note that, in general, direct manipulation with the network components is not recommended for it may destroy the object integrity.

Functions `cnNumNodes` and `cnNodes` return the number and the list of names, respectively, of network nodes. For each node of a network, for example `cnet1`, one can obtain its parents by calling `cnParents` function or find its children through `cnEdges` function.

```
> cnNumNodes(cnet1)

[1] 4

> cnNodes(cnet1)

[1] "N1" "N2" "N3" "N4"

> cnEdges(cnet1)
```

```
$N2
[1] "N3" "N4"

$N3
[1] "N4"
```

```
> cnParents(cnet1)

$N3
[1] "N2"

$N4
[1] "N2" "N3"
```

In addition, the corresponding `cnMatParents` and `cnMatEdges` functions return matrices instead of lists. Specifically, `cnMatParents` returns a binary matrix representing the pairwise node connections, and it is especially useful for comparing networks with the same number of nodes. Alternatively, `cnMatEdges` returns a two-column matrix of ordered pairs encoding the edges with direction from the first to the second.

```
> cnMatParents(cnet1)

   N1 N2 N3 N4
N1  0  0  0  0
N2  0  0  0  0
N3  0  1  0  0
N4  0  1  1  0
```

```
> cnMatEdges(cnet1)

     [,1] [,2]
[1,] "N2" "N3"
[2,] "N2" "N4"
[3,] "N3" "N4"
```

`cnProb` function provides an access to the complete probability table of a network, which is a recursive list and can be quite large for networks with big parent sets and many categories. Conditional probabilities are reported in the following format. First, node name and its parents are given, then a list of probability values corresponding to all combinations of parent categories (put in brackets) and node categories. In the following example the first node ($N1$) has two categories ($C1$ and $C2$) and no parents, thus two numbers are given, probability 0.68 for the first category and 0.32 for the second. The third node ($N3$) has two categories and one parent ($N2$) and consequently two pairs of probabilities are reported , one for $N2 = C1$ and another for $N2 = C2$.

```
> cnProb(cnet1)

Node[N1], Parents:
[]C1  0.68
[]C2  0.32
 Node[N2], Parents:
[]C1  0.55
[]C2  0.45
 Node[N3], Parents: N2
```

```
[ C1]C1  0.12
[ C1]C2  0.88
[ C2]C1  0.84
[ C2]C2  0.16
 Node[N4], Parents: N2 N3
[ C1 C1]C1  0.26
[ C1 C1]C2  0.74
[ C1 C2]C1  0.57
[ C1 C2]C2  0.43
[ C2 C1]C1  0.4
[ C2 C1]C2  0.6
[ C2 C2]C1  0.49
[ C2 C2]C2  0.51
```

An important characteristic of a `catNetwork` is its complexity. The complexity is an integer number specifying the number of parameters needed to define the probability structure of the network. For example, the complexity of a network with nodes without parents and two categories each equals the number of nodes. The complexity of a network object can be obtained by calling `cnComplexity` function.

```
> cnComplexity(cnet1)

[1] 8
```

The complexity is a key property of a network, decisive in the network estimation and model selection problems.

2.1 Drawing a Network

Catnet provides several alternatives for visualizing a network. They are implemented in the function `cnPlot`. If the *igraph* package is installed, a `catNetwork` object will be coerced to a `igraph` object and plotted. Alternatively, `cnPlot` may generate a dot-file, compatible with the external Graphviz software package ([11]). Dot-files can be rendered to a postscript or a pdf files using the `dot` executable from Graphviz or directly visualized by `dotty` or `tcldot`. There is an auxiliary to `cnPlot` function called `cnDot` that generates and saves dot-files specifically.

```
cnPlot(cnet1)
cnDot(cnet1, "cnet1.dot")
```

2.2 Topological Order

Since any `catNetwork` is a DAG, there is a natural order of its nodes, such that any node has parents only among the nodes appearing earlier in the order. In fact, a network may have many compatible orders and the function `cnOrder` returns just one of them. `cnOrder` takes as an input either a `catNetwork` object or a list of parent sets. The next example illustrates its usage.

```
> cnOrder(cnet1)

[1] 1 2 3 4

> cnOrder(cnet1@parents)

[1] 1 2 3 4
```

The topological order is important in the context of network learning and it is a key element in the *catnet*'s search methodology.

2.3 Equivalent Graph Representation

An important result from the theory of Bayesian networks states that all networks with common sets of nodes can be organized in equivalent classes. According to definition, for any two equivalent networks there are probability structures such that their joint probabilities are equal. More on the topic one can find in [24] and [6]. Function `dag2cpdag` generates the Complete Partially Directed Graph (CPDAG) for a network according to the algorithms given in [6]. Note that in a CPDAG some edges are not directed to reflect the freedom of choosing directions without leaving the corresponding equivalent class.

```
> set.seed(456)
> cnet2 <- cnRandomCatnet(numnodes = 10, maxParents = 3, numCategories = 2)
> cnEdges(cnet2)

$N2
[1] "N8"  "N10"

$N4
[1] "N3"

$N6
[1] "N8"

$N8
[1] "N5"

> pcnet2 <- dag2cpdag(cnet2)

add  N3 -> N4
add  N5 -> N8
add  N10 -> N2

> cnEdges(pcnet2)

$N2
[1] "N8"  "N10"

$N3
[1] "N4"

$N4
[1] "N3"

$N5
[1] "N8"

$N6
[1] "N8"

$N8
[1] "N5"

$N10
[1] "N2"
```

2.4 Comparing Networks

Often, one has two networks with the same nodes and wants to evaluate the difference between them. There are two basic criteria for comparing networks. First, a topological one that compares the graphical structure of the networks and second, a probabilistic one, in regard to the distributions specified by the networks. *Catnet* employs several measures for topological comparison such as the number of true positive/false positive/false negative directed edges, the parent Hamming distance - the number of different elements between the corresponding parent matrices, the number of true positive/false positive/false negative undirected edges (skeleton), and the number of false positive/negative Markov pairs (pairs that have common descendants). Also included is a measure comparing the node order in the networks. The user can compare two networks by calling `cnCompare` function. It returns a a `catNetworkDistance` object containing the values of the provided measures. More details can be found in the help file of `cnCompare`.

```
> cnet3 <- cnRandomCatnet(cnNumNodes(cnet2), maxParents = 2, numCategories = 2)
> cnet3@nodes <- cnet2@nodes
> cnCompare(object1 = cnet2, object2 = cnet3)

 Edges:
                   TP = 0,
                   FP = 2,
                   FN = 5,

 Hamming:
              (FP+FN) = 7,
                  exp = 9,

 Skeleton:
                   TP = 0,
                   FP = 2,
                   FN = 5,

 Order:
                   FP = 0,
                   FN = 0,

 Markov blanket:
                   FP = 1,
                   FN = 1
```

3 Generating Samples from Network and Making Predictions

In addition to the row-sample data representation as often used in statistical practice, *catnet* also allows a column-sample format, the later being a standard for storing micro-array data. In the latter case, the samples are organized in columns and each row represents a node. The package provides two function for data generation, `cnSample` and `cnSamplePert`. The second one allows the user to generate a perturbed sample, a sample with some of its nodes having fixed, non-random, values. In micro-biology, the perturbation techniques is an important tool for inferring causal relationships in regulatory networks.

In the following example we generate a random sample of size 100 from `cnet1` object that have been created earlier. By setting the `output` argument to be "matrix", we obtain a result in `matrix` form. Alternatively, a sample as `data.frame` can be generated.

```
> samples1 <- cnSamples(object = cnet1, numsamples = 100, output = "matrix")
> dim(samples1)

[1]   4 100

> samples1 <- cnSamples(object = cnet1, numsamples = 100, output = "frame")
> dim(samples1)

[1] 100   4
```

Another possibility is to generate perturbed samples with fixed values at particular nodes. For the purpose, the user may call `cnSamplesPert` function and specify an additional vector argument, `perturbations`, of length the number of nodes and values either fixed categorical indices or zero to mark the random nodes which values has to be sampled. In the next example a sample of size 10 is generated with random first two nodes but perturbed third and fourth nodes that take their first and second category, respectively.

```
> samples2 <- cnSamplesPert(object = cnet1, perturbations = c(0,
+     0, 1, 2), numsamples = 10)
```

For prediction purposes one can use `cnPredict` function with parameters network object and data. In the data, only the node positions marked as not-available (NA) are predicted. The nodes are assigned categorical values based on the maximum probability criterion. If for example, given a particular instance of its parenthood, a node has three categories with probabilities (0.2, 0.5, 0.3), then the second category will be assigned as its value.

```
> numnodes <- cnNumNodes(cnet2)
> samples3 <- cnSamples(object = cnet2, numsamples = 12, output = "matrix")
> samples3[numnodes - 2, ] <- rep(NA, 12)
> samples3[numnodes - 1, ] <- rep(NA, 12)
> samples3[numnodes, ] <- rep(NA, 12)
> newsamples <- cnPredict(object = cnet2, data = samples3)
```

4 Learning Network form Data

In contrast to some existing network learning algorithms such as Grow-Shrink, Incremental Assosiation ([25], [23]) and Hill-Climbing ([8]), to mention a few, *catnet* does not 'learn' but rather searches for networks according to the MLE criterion. For a known order of the nodes, the package provides a function (`cnSearchOrder`) that finds the exact (up to equivalence) MLE solution of the problem of fitting network to data. When the node order is unknown, *catnet* provides appropriate stochastic search functions (`cnSearchSA`, `cnSearchSAcluster`).

4.1 Search for Given Order

As put forward in [7], the model search in the space of all networks can be restricted to a smaller space of networks that are consistent with a particular node ordering. The spaces of networks with fixed node order are not only smaller but more 'regular'. In [14], authors evaluate the regularity in terms of posterior distribution of features and also conclude that a search restricted to a particular node ordering does not necessarily preclude good network recovering. Empirical studies also confirm that on order restricted network spaces, the likelihood functions have less local variability, thus discontinuity, which facilitates likelihood based estimations.

Function `cnSearchOrder` is the main computational tool provided by *catnet*. The function implements a Dynamic Programming (DP) algorithm for searching in the space of networks having a node

order specified by the user. The result is a set of networks with increasing complexity up to a given maximum value. In other words, each resulting network is exact the Maximum Likelihood Estimator (MLE) for the corresponding complexity. The user may proceed by selecting a particular network from this set of networks by applying a model selection criterion. In this regard, functions `cnFindAIC` and `cnFindBIC` mentioned below can be useful.

```
> set.seed(789)
> cnet2 <- cnRandomCatnet(numnodes = 10, maxParents = 2, numCategories = 2)
> nodeOrder <- order(runif(cnNumNodes(cnet2)))
> cnet2

A catNetwork object with  10  nodes,  2  parents,  2  categories,
 Likelihood =  0 , Complexity =  11 .

> samples <- cnSamples(object = cnet2, numsamples = 100, output = "frame")
> netlist <- cnSearchOrder(data = samples, perturbations = NULL,
+     maxParentSet = 2, maxComplexity = 20, nodeOrder, parentsPool = NULL,
+     fixedParentsPool = NULL)
> bnet <- cnFind(netlist, 20)
> bnet

A catNetwork object with  10  nodes,  2  parents,  2  categories,
 Likelihood =  -624.0839 , Complexity =  20 .
```

`cnSearchOrder` has two mandatory parameters: `data` and `maxParentSet`. The categorical structure is inferred from the data. If `maxComplexity` is not specified, the search is applied to the maximum possible complexity.

In many practical problems some prior information about the network structure is available which the user may want to include in the search process. Such prior information can be obtained from experts in the field of interest or to be based on preceding research. In all of its search functions, *catnet* includes the parameters `parentsPool` and `fixedParentsPool`, that can be used for specifying edge constrains.

The first parameter, `parentsPool`, specifies a set of possible parents for each node and in this way, some nodes can be excluded as potential parents. Additionally, the `fixedParentsPool` parameter specifies edge inclusion rules - the user can give some nodes mandatory set of parents. By manipulating these two parameters, a variety of edge constrains can be implemented.

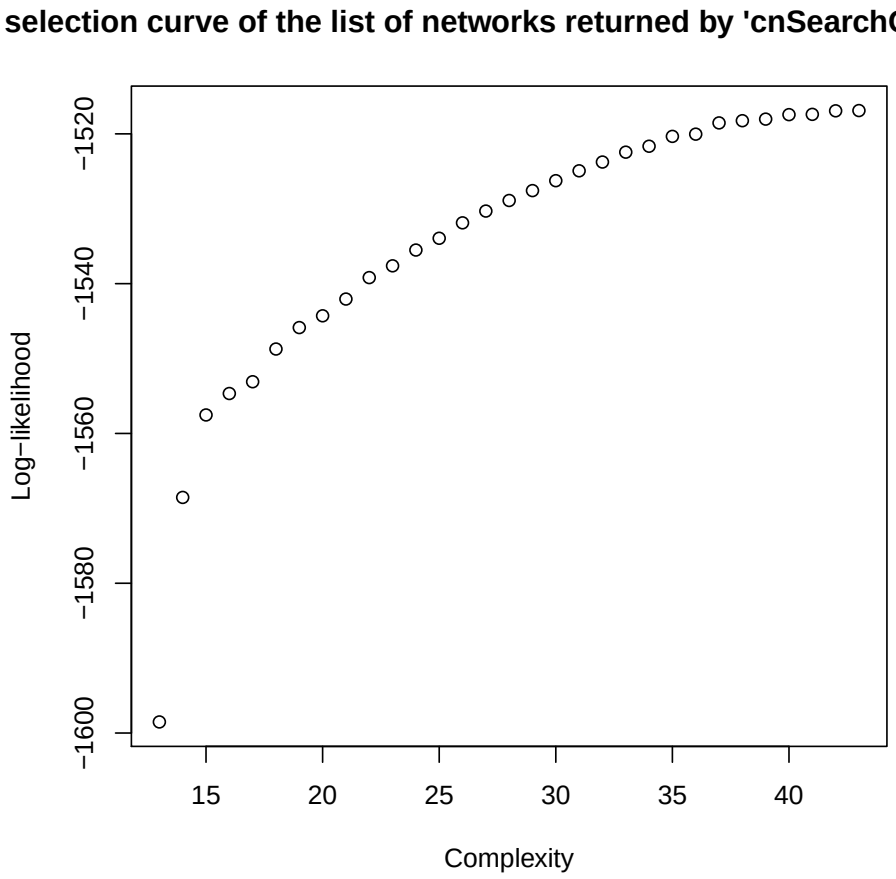
In the next example, we generate a random network with 12 nodes and then search for the best fitting networks that comply with the following requirements: (1) the last node is not a parent to anyone else, and (2) the first two nodes are necessarily parents to all of the rest nodes. The search is restricted to the 'true' order, obtained by `cnOrder(cnet)`, of the nodes in the simulated data.

Finally, we draw the Log-likelihood vs. complexity curve for the resultant list of networks.

```
> set.seed(123)
> nnodes <- 12
> cnet <- cnRandomCatnet(numnodes = nnodes, maxParents = 5, numCategories = 2)
> norder <- cnOrder(cnet)
> parPool <- vector("list", nnodes)
> for (i in 1:nnodes) parPool[[i]] <- 1:(nnodes - 1)
> fixparPool <- vector("list", nnodes)
> for (i in 3:nnodes) fixparPool[[i]] <- c(1, 2)
> samples <- cnSamples(cnet, 200)
> eval <- cnSearchOrder(data = samples, perturbations = NULL, maxParentSet = 2,
+     maxComplexity = 200, nodeOrder = norder, parentsPool = parPool,
+     fixedParentsPool = fixparPool)
> eval
```

```
 Number of nodes    = 12,
 Sample size        = 200,
 Number of networks = 31
 Processing time    = 0.043
```

```
> plot(eval@complexity, eval@loglik, xlab = "Complexity", ylab = "Log-likelihood",
+     main = "Model selection curve of the list of networks returned by 'cnSearchOrder' function")
```



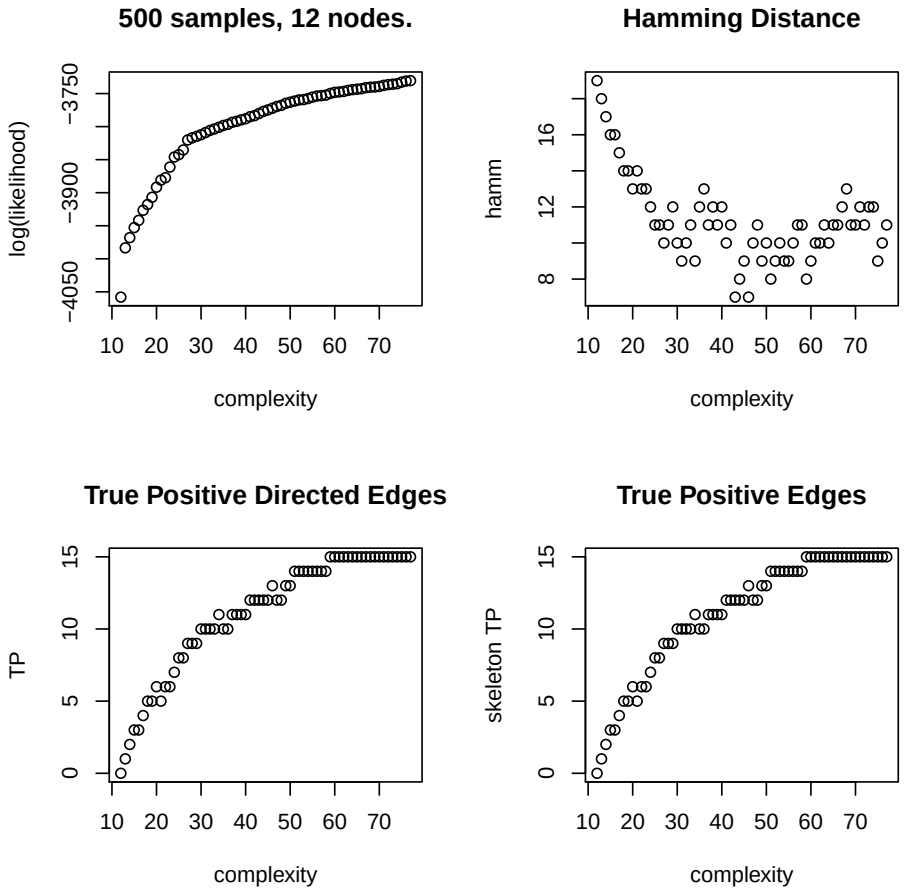
4.2 Evaluation

For search validation purposes, *catnet* provides the function `cnEvaluate`. For given `catNetwork` object and some data, the function finds a set of optimal networks that maximize the likelihood of that data and have complexity in the range starting with the minimum possible complexity up to a value specified by the user. Internally, the function calls `cnSearchOrder` to search in the space of networks having the same node order as the given network object. Moreover, the resulting optimal networks are compared to the given network using several criteria (see the manual of `cnCompare` for more details). The output of `cnEvaluate` is a `catNetworkEvaluate` object that contains all relevant diagnostic information and which can be conveniently plotted.

```
> numsamples <- 500
> samples <- cnSamples(object = cnet, numsamples, output = "frame")
```

```
> maxComplexity <- cnComplexity(cnet)
> perturbations <- matrix(rep(0, cnet@numnodes * numsamples), nrow = cnet@numnodes)
> netlist <- cnEvaluate(object = cnet, data = samples, perturbations = perturbations,
+     maxComplexity)
> cnPlot(netlist)
> bnet <- cnFind(netlist, maxComplexity)
> bnet

A catNetwork object with  12  nodes,  4  parents,  2  categories,
 Likelihood =  -3778.162 , Complexity =  77 .

> cnCompare(cnet, bnet)

 Edges:
                  TP = 18,
                  FP = 6,
                  FN = 1,

 Hamming:
             (FP+FN) = 7,
                 exp = 33,

 Skeleton:
                  TP = 18,
                  FP = 6,
                  FN = 1,

 Order:
                  FP = 0,
                  FN = 0,

 Markov blanket:
                  FP = 6,
                  FN = 4

> bnet2 <- cnFind(netlist, maxComplexity/2)
> bnet2

A catNetwork object with  12  nodes,  3  parents,  2  categories,
 Likelihood =  -3829.37 , Complexity =  39 .

> cnCompare(cnet, bnet2)

 Edges:
                  TP = 13,
                  FP = 1,
                  FN = 6,

 Hamming:
             (FP+FN) = 7,
                 exp = 24,

 Skeleton:
```

```
                TP = 13,
                FP = 1,
                FN = 6,

Order:
                FP = 0,
                FN = 0,

Markov blanket:
                FP = 0,
                FN = 13
```



Another application of `cnEvaluate` function might be reducing a network to a less complex one. A stochastic way to perform this is to generate a large sample from the original network, then evaluate the sample to obtain a list of networks with complexity up to the target one. In the above example, `bnet2` is a reduced version of the original network `cnet` with about twice smaller complexity.

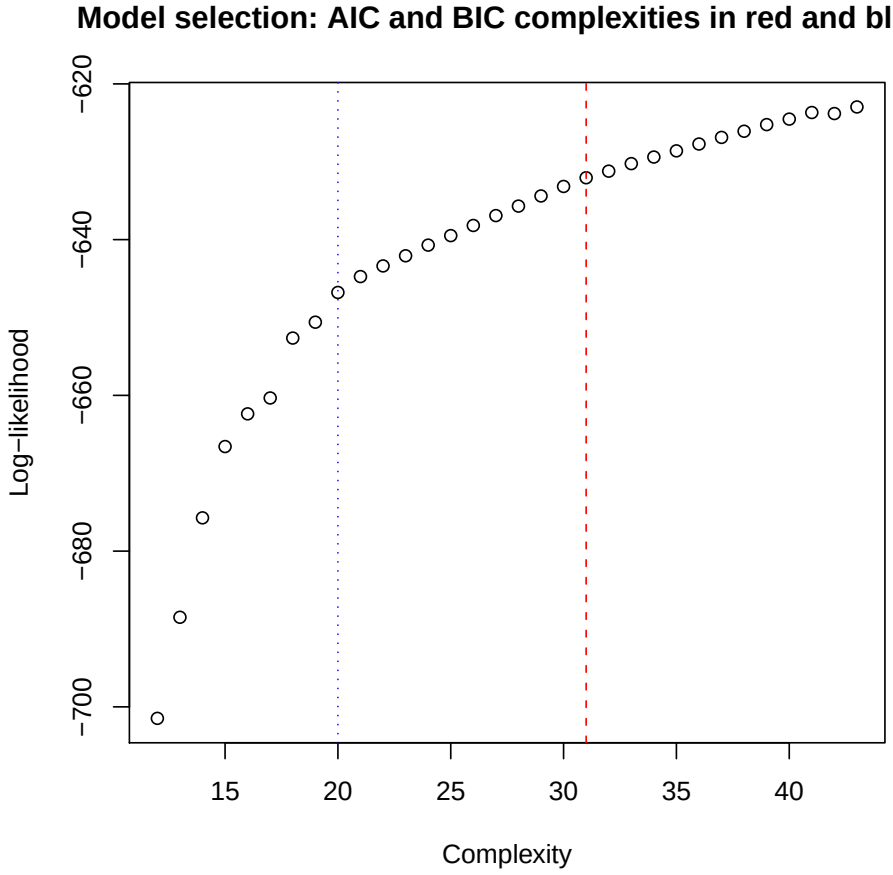
4.3 Model Selection with AIC and BIC criteria

The model selection problem in the context of network learning is one of the focuses of *catnet*. As we have mentioned earlier, the package aims to provide flexibility and allows the user to select a network

from a list of optimal ones according to his/her needs. Methodological details behind the model selection procedures implemented in *catnet* can be found in [19].

Recall that by calling one of the functions `cnSearchOrder`, `cnSearchSA` and `cnSearchSAcluster`, one obtains a list of networks, more precisely `catNetwork` objects, such that each network has a unique complexity. From this list one may select a network based on particular criterion such as AIC or BIC. In the next example both AIC and BIC selections are made and their complexity is marked on the model selection curve.

```
> set.seed(345)
> cnet6 <- cnRandomCatnet(numnodes = 12, maxParents = 5, numCategories = 2)
> samples <- cnSamples(object = cnet6, numsamples = 100, "matrix")
> eval <- cnSearchOrder(data = samples, perturbations = NULL, maxParentSet = 2,
+     maxComplexity = 0, nodeOrder = order(runif(1:dim(samples)[1])),
+     parentsPool = NULL, fixedParentsPool = NULL, echo = FALSE)
> anet <- cnFindAIC(object = eval)
> anet

A catNetwork object with  12  nodes,  2  parents,  2  categories,
 Likelihood =  -632.0516 , Complexity =  31 .

> bnet <- cnFindBIC(object = eval, numsamples = dim(samples)[2])
> bnet

A catNetwork object with  12  nodes,  2  parents,  2  categories,
 Likelihood =  -646.7812 , Complexity =  20 .

> plot(eval@complexity, eval@loglik, xlab = "Complexity", ylab = "Log-likelihood",
+     main = "Model selection: AIC and BIC complexities in red and blue.")
> abline(v = anet@complexity, lty = 2, col = "red")
> abline(v = bnet@complexity, lty = 3, col = "blue")
```



4.4 Stochastic Search in the Space of Orders

In cases when prior knowledge about the order of the nodes is not available, the user can employ a stochastic search method in the space of all possible node orders. In a way, this will break the general search problem into two smaller ones - 'order search' and 'search in order'. The space of orders includes all possible node permutations and, although still huge, is somehow smaller than that of all possible networks. This approach of order factorization is also implemented in [14] but in a different, Bayesian, context. The idea of considering order learning as a sub-problem is also followed in [15] and [1] with constraint-based algorithms for conditional independence. What makes *catnet*'s approach different is the strictly likelihood based, non-Bayesian, learning framework.

`Catnet` package provides two stochastic search functions: `cnSearchSA` and `cnSearchSAcluster`. These functions implement a Simulated Annealing algorithm. More technical details are given below.

Let $\mathcal{A} = \{x_i\}_{i=1}^n$ be a set of n discrete variables with number of categories $\mathcal{C} = \{c_i\}_{i=1}^n$. A categorical network $\mathcal{G}$ with nodes the variables of $\mathcal{A}$ can be described by its edge structure and conditional probabilities. The edges can be encoded by a binary matrix $E = \{E_{ij}\}_{i,j=1}^n$ such that $E_{ij} = 1$ if x_i is a parent of x_j and zero otherwise. Let $\mathcal{X} = \{X^j\}_{j=1}^k$ be a k-sample data from an unknown network with nodes $\mathcal{A}$ and number of categories $\mathcal{C}$. Then the likelihood of $\mathcal{X}$ with respect to $\mathcal{G}$ is

$$\mathcal{L}(\mathcal{G}|\mathcal{X}) = \prod_{i=1}^{n} \prod_{j=1}^{k} P(X_i^j | \Pi_i(X^j)), \tag{1}$$

where $\Pi_i(X_j)$ is the realization of the parent set of x_i in $\mathcal{N}$ for the j-th sample X^j. For an order Ω of the nodes $\mathcal{A}$, we define $\hat{\mathcal{N}}(\Omega, \mathcal{X})$ to be the network consistent with Ω that maximizes the likelihood (1). Define

$$P(\Omega) \propto \mathcal{L}(\hat{\mathcal{N}}(\Omega, \mathcal{X})|\mathcal{X}).$$

In fact, $\hat{\mathcal{N}}$ may be only one representative network from the set of equivalent networks maximizing the likelihood (1).

The functions `cnSearchSA` and `cnSearchSAcluster` implement a Metropolis algorithm with acceptance probability

$$\pi(\Omega_2|\Omega_1, \beta) \propto 1_{\Omega_2 \in \mathcal{S}(\Omega_1)} exp(-[\mathcal{L}(\hat{\mathcal{N}}(\Omega_2, \mathcal{X})) - \mathcal{L}(\hat{\mathcal{N}}(\Omega_1, \mathcal{X}))]^+/\beta), \tag{2}$$

where $\mathcal{S}(\Omega_1)$ is a neighborhood of Ω_1 and $\beta > 0$ is a parameter which specifies the temperature for the Simulating Annealing. The parameter β gradually decreases according to a cooling schedule specified by the function's parameters. The neighborhood $\mathcal{S}(\Omega)$ for an order Ω includes all orders obtained from Ω by picking up randomly a node index and exchange places with the next node index in Ω, eventually performing this operation given number of times as specified by `orderShuffles` parameter. In this way one can control the extend of $\mathcal{S}(\Omega)$. A large variety of search scenarios can be implemented by varying function parameters and the user should choose one suitable to his/her application needs.

In addition to the parameters of `cnSearchOrder`, `cnSearch` accepts also

1. `tempStart` - the starting temperature of the annealing process
2. `tempCoolFact` - the cooling factor from one temperature step to another. It is a number between 0 and 1, inclusively. For example, if `tempStart` is the temperature in the first step, `tempStart*tempCoolFact` will be temperature in the second
3. `tempCheckOrders` - the number of proposals to be checked, or with other words, order selections from the current order's neighborhood, at each step before decreasing the temperature. Thus, if Ω is the current order at some temperature, totally `tempCheckOrders` orders from $\mathcal{S}(\Omega)$ will be proposed and accepted or rejected, before factoring down the temperature.
4. `maxIter` - the maximum number of orders to be checked. If for example `maxIter` is 40 and `tempCheckOrders` is 4, then 10 temperature decreasing steps will be eventually performed.
5. `orderShuffles` - a number that controls the extend of $\mathcal{S}(\Omega)$, the neighborhood of order Ω. The element of $\mathcal{S}(\Omega)$ are obtained from Ω by `orderShuffles` switches of two node indices.
6. `stopDiff` - a stopping criterion. If at a current temperature, after `tempCheckOrders` orders being checked, no likelihood improvement of level at least `stopDiff` is detected, then the SA stops and the function exists. Setting this parameter to zero guarantees exhausting all of the maximum allowed `maxIter` order searches.
7. `priorSearch` - a result from previous search. This parameters allows a new search to be initiated from the best order found so far. Thus a chain of searches can be constructed with varying parameters providing greater adaptability and user control.

The set of parameters of `cnSearchSA` function allows implementing a variety of search strategies. Unfortunately, it is hard to recommend a default set of parameters good in many situations. We can only make the following suggestion to the user: try several parameter combinations, perform repeated search with limited number of iterations (not too large `maxIter`) with each of them, choose a setting with the most consistent results (in terms of likelihood for a fixed complexity) and perform a longer search with the already chosen set of parameters. Another hint is to look at the acceptance rate of the SA algorithm and choose a setting that gives about 10 to 30 percent acceptance. In any case, a number of runs of `cnSearchSA` are recommended before making conclusions, a general recommendation for any MCMC procedure.

```
> netlist <- cnSearchSA(data = samples, perturbations = NULL, maxParentSet = 2,
+     maxComplexity = 20, parentsPool = NULL, fixedParentsPool = NULL,
+     tempStart = 1, tempCoolFact = 0.9, tempCheckOrders = 4, maxIter = 40,
+     orderShuffles = 1, stopDiff = 1e-04, priorSearch = NULL)
> bnet <- cnFind(netlist@nets, cnComplexity(cnet2))
> bnet

A catNetwork object with  12  nodes,  0  parents,  2  categories,
 Likelihood =  -701.4731 , Complexity =  12 .
```

The function `cnSearchSA` has a parameter called `priorSearch`, which can take the result of previous call to `cnSearchSA`. In that case the new search starts where the previous search ended thus trying to improve upon the best set of networks that have been already found. This mechanism allows implementation of sophisticated multi-stage search scenaria by adapting to the user needs.

4.5 Parallel Processing

By its nature, the search for best network according to a likelihood based score is NP-Complete, thus very hard, problem (see [5]). Inherently, the search functions implemented by *catnet* are highly intensive computationally and some means for processing in cluster environment are necessary.

Currently, *catnet* depends on *snow* package, [22], for providing parallel processing. In this way, multi-core/threads hardware abilities can be employed for faster stochastic search using `cnSearchSAcluster`, the cluster processing enabled analog of `cnSearchSA`. `cnSearchSAcluster` function takes as additional parameters the number of cluster units to be employed and their host ('localhost' as default), either as a name or IP address. `cnSearchSAcluster` employs the specified parallel processing units by running an independent SA search in each of them, as performed by `cnSearchSA` function, and finally chooses the best achieved result. More details are available in the manual pages.

The next example demonstrates a Simulated Annealing search using two parallel processing units running on the local machine.

```
> set.seed(345)
> bsnow <- FALSE
> try(bsnow <- require(snow), silent = TRUE)
> if (bsnow) {
+     cnet7 <- cnRandomCatnet(numnodes = 12, maxParents = 5, numCategories = 2)
+     samples <- cnSamples(object = cnet7, numsamples = 100)
+     netlist2 <- cnSearchSAcluster(data = samples, perturbations = NULL,
+         maxParentSet = 2, maxComplexity = 0, parentsPool = NULL,
+         fixedParentsPool = NULL, selectMode = "BIC", tempStart = 1,
+         tempCoolFact = 0.9, tempCheckOrders = 8, maxIter = 40,
+         orderShuffles = 1, stopDiff = 1e-04, clusterNodes = 2,
+         clusterHost = "localhost")
+     bnet <- cnFind(netlist2@nets, cnComplexity(cnet6))
+     bnet
+ }

A catNetwork object with  12  nodes,  2  parents,  2  categories,
 Likelihood =  -627.1141 , Complexity =  43 .
```

5 Conclusion

The *catnet* package is intended to diversify the existing tools for graphic statistical modeling by implementing a complete inference set for discrete Bayesian networks. Covering all potential application areas

of the package is out of the scope of this introduction. The user, however, is referred to the "Getting Started" vignette and the demonstrations accompanying the package for some application ideas.

References

[1] Acid, S., Campos, L., Huete, J., The Search of Causal Orderings: A Short Cut for Learning Belief Networks. 2001, In Proc. 8-th conference on Uncertainty in Artificial Intelligence.

[2] Beinlich, I., Suermondth, G., Chavez, R., Cooper, G., The ALARM monitoring system. 1989, In Proc. 2-nd Euro. Conf. on AI and Medicine.

[3] Bouckaert, R. Optimizing causal orderings for generating dags from data. 1992, In Proc. Conf. on Uncertainty in ArtiïňĄcial Intelligence, pages 9-16. Morgan-Kaufmann.

[4] Buntine, W., A guide to the literature on learning probabilistic networks from data. 1996, IEEE Transaction on Knowledge and Data Engineering, 8:195-210.

[5] Chickering, D. M., Learning Bayesian Networks is NP-Complete. 1996(a), In D. Fisher, H. Lenz (Eds.) Learning from Data: Artificial Intelligence and Statistics V, Ch. 12, 121-130.

[6] Chickering, D. M., Learning Equivalence Classes of Bayesian-Network Structures. 1996(b), Journal of Machine Learning research, 2, pp. 445-498.

[7] Cooper, G. F., Herskovitz, E., A Bayesian method for the induction of probabilistic networks from data. 1992, Mach. Learn., 9, pp. 309-347

[8] Daly, R., Shen, Q., Methods to Accelerate the Learning of Bayesian Network Structures. 2007, In Proc. of the UK Workshop on Computational Intelligence, Imperial College, London.

[9] Heckeman, D., Geiger, D., Chickering, D., Learning Bayesian networks: The combination of knowledge and statistical data. 1995, Machine Learning, 20(3), pp. 197-243.

[10] Gentleman, R., Whalen, E., Huber, W., Falcon, S., graph: A package to handle graph data structures. R package. 2009, package version 1.24.1

[11] Graphviz - Graph Visualization Software http://www.graphviz.org/

[12] Friedman, N., Goldszmidt, M., Wyner, A., Data Analysis with Bayesian Networks: A Bootstrap Approach. 1999, Proc. Fifteenth Conf. on Uncertainty in Artificial Intelligence (UAI).

[13] Friedman, N., Goldszmidt, M., Wyner, A., Using Bayesian Networks to Analyze Expression Data. 2000, Journal of Computational Biology, 7, pp. 601-620.

[14] Friedman, N., Koller, D., Being Bayesian about network structure: A Bayesian approach to structure discovery in Bayesian networks. 2003, Mach. Learn., 50, pp. 95-125.

[15] Larranaga, P. , Kuijpers, C., Murga, R., Yurramendi, Y., Learning Bayesian network structures by searching for the best ordering with genetic algorithms. 1996, IEEE Trans. Pattern Anal. and Mach. Intell.,26:487-493.

[16] Neapolitan, R., E., Learning Bayesian Networks. 2003, Prentice Hall.

[17] Pearl, J., Verma, T.S., A theory of inferred causation. 1991, 2-nd Conference on the Principles of Knowledge Representation and Reasoning, Cambridge, MA.

[18] Pearl, J., Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. 1988, Morgan Kaufmann.

[19] Salzman, P., Almudevar, A., Using Complexity for the Estimation of Bayesian Networks. 2006, Statistical Applications in Genetics and Molecular Biology, Vol. 5, Issue 1.

[20] Scutari, M., bnlearn: Bayesian network structure learning. R package. 2010, package version 1.8

[21] Sebastiani, P., Abad, M., Ramoni, M., Bayesian Networks for Genomic Analysis. 2004, Genomic Signal Processing and Statistics, pp. 281-320.

[22] Tierney, L., Rossini, A. J., Na Li, Sevcikova, H., snow: Simple Network of Workstations. R package. 2008, package version 0.3-3

[23] Tsamardinos, I., Aliferis, C., Statnikov, A., Algorithms for Large Scale Markov Blanket Discovery. 2003, In Proc. of the 16-th Inter. Florida Artificial Intelligence Research Society Conference, pp. 376-381, AAAI Press.

[24] Verma, T., Pearl, J., Equivalence and synthesis of causal models. 1990, In Henrion,M., Shachter, R., Kanal, L., and Lemmer, J., editors, Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence, pp. 220-227.

[25] Yaramakala, S., Margaritis, D., Speculative Markov Blanket Discovery for Optimal Feature Selection. 2005, In ICDM'05, Proceedings of the 5-th IEEE Conference on Data Mining, pp. 809-812.