MCMCBNE package v1.0 manual

Sangkyu Lee
Ph.D. candidate
Medical Physics Unit, McGill University
Montreal,Quebec, Canada

August 16, 2015

1 Package highlights

Markov Chain Monte Carlo sampling for Bayesian Network Ensemble (MCMCBNE) is a Matlab package for learning/testing an ensemble of Bayesian Network (BN) graphs on multivariate data. The main motivation was to develop a BN-based classifier for predicting radiotherapy response using longitudinal biomarker measurements and radiotherapy plan/ other patient-specific information. However, the toolkit can be used for other classification problems after customizing a data import component. It was developed as an extension to the Bayes Net Toolbox (https://github.com/bayesnet/bnt) which provides backbone functions for Bayesian Network graph and parameter training. Major additional features implemented by the MCMCBNE include:

- Number of input variables can be reduced by two types of filtering schemes: Koller-Sahami/ L1 regulated logistic regression.
- MCMC graph sampling can be guided by the two types of user-defined priors: causality (reject the edges in non-causal direction) and biological (edges reported in literatures are given higher prior probability)
- Ensemble of Bayesian networks can be formed to be used for classification.
- Classification performance can be measured in a cross-validation/0.632+ bootstrap settings.
- Codes for graph learning and performance testing are parallelized for submission to high-computing clusters.
- Also included are some visualization codes useful for reviewing and evaluating Bayesian Network models.

Organization of the package is shown in figure 1.

2 Dependencies

The following packages (all Matlab based) need to be installed beforehand:

- Bayes Net Toolbox (BNT) (https://github.com/bayesnet/bnt)

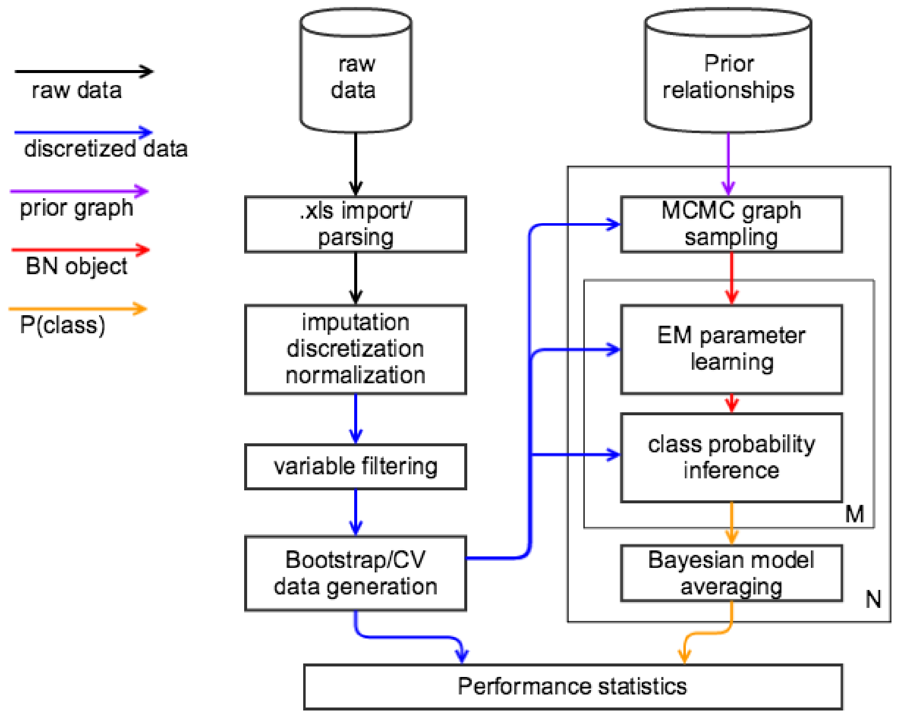

Figure 1: Schematic diagram of the modules constituting the MCMCBNE package. M: number of graphs in one ensemble, N: number of bootstrap/CV datasets

- Dose Response Explorer (DREES) (https://github.com/yw2026/DREES)
- Statistics and Bioinformatics toolbox from MATLAB

3 Data Import

The data importing module was written according the current convention of radiotherapy record keeping. For other uses, users are encouraged to write their own module that suits the structure of a specific dataset. In this case, jump to the section for a graph learning module.

The module reads biological and physical/clinical data from separate .xls files. The spreadsheet should be stored in the Excel 97 version with the first row indicating variable names. Each spreadsheet is expected to be organized in its own specific ways:

- The physical/clinical data spreadsheet is organized in a rows-for-instances and columns-for-variables convention. It is expected to contain a variable to be used as a class. The name of the variable, specified at a variable class_name in a file kyu_BN_readdata.m, should match the name as written in the spreadsheet. The code also recognizes a column named as 'include' which can be included in the spreadsheet should a user filter instances individually. The value for the 'include' is set to be 0 for the patients to be excluded from the imported data, or otherwise 1. By default (e.g. there is no such column as 'included'), the instances with a known class value are imported.
- The biological data sheet, however, has each row for the measurement from one patient at one time point. A row identifier is written in the following format: "PatientID-time point" (ex:L012-3). One-digit number from 1 to 5 is assigned to the following time point, which differs for 2 fractionation groups due to the current data collection protocol (table 1):

There are three pre-processing steps applied to the raw data imported from spreadsheets:

1. Imputation: missing entries are filled in, using one of the two options:
    - median: fill the missing value with the median of the existing data

Table 1: Assignment of time point labels for reading biomarker data spreadsheet.

time label	conventional	hypo fraction
1	baseline	baseline
2	mid-treatment	end-treatment
3	end-treatment	3-month postRT
4	3-month postRT	6-month postRT
5	6-month postRT	

- k-nearest neighbor: Similarity between patients is evaluated using the variables with no NaN. The missing entry is filled by the value of the patient with the highest similarity.

2. Normalization

   - standard z-score: data is shifted/scaled to the zero mean and unitary standard deviation.
   - min-max: data is linearly transformed so that the smallest and the largest entry hold the value of 0 and 1 respectively.
   - softmax: similarly to above, data is squeezed to the [0 1] range, but a hyper tangent function is used instead of a linear one for the mapping. This reduces the effect of outliers or extreme values on the normalization result.

3. Discretization

   - maximum mutual information: bin boundary is optimized to maximize mutual information with respect to a class variable.
   - k-means: clustering-based unsupervised discretization option.

These options can be set in the file kyu_BN_readdata.m.

The import module can be called in the Matlab command window as follows:

```
>> SBRTfilter = 2; % select 2 Gy per fraction pts only.
>> disc = 3; % use K-means discretization
>> [data,data_c,data_missing,data_c_missing,...
    labels,mi,studyid,FracSize,COMSI] = ...
    kyu_BN_readdata(SBRTfilter,disc)
The following pts were removed due to missing class:
    'W005'
    'W010'
...
The following variables are found:
1. a2m_pre
2. a2m_intra
3. a2m_end
...
choose the variables, separated by comma: 1,4
--------------odds ratio of the selected variables-----------------
1. a2m_pre: x.xx (y.yy, z.zz)
...
(event rate: 00.00 %)
```

Calling the function will first show the IDs of the removed instances due to a missing class value, followed by a user prompt first showing the choices for the variables detected in the raw data spreadsheet. Specify the variables that you want to include into a data matrix by a comma separated list of numbers as shown in the list. Finally, it shows the computation results for the odds ratios of the selected variables. The 95% confidence interval of the odds is shown in a parenthesis. Also shown is a rate of "events" which is defined as the fraction of class==2.

The required input parameters for kyu_BN_readdata.m are:

- SBRTfilter: determines which fractionation group to select.

  1: every patients,

  2: conventional fractionation (dose per fraction $\leq$ 2 Gy)

  3: hypo+SBRT (dose per fraction $\geq$ 3 Gy)

- disc: discretization option

  2. maximum mutual information

  3. k-means

The output of the functions includes the data matrices that are pre-processed in different ways:

- data: discretized/ imputed data
- data_c: continuous(normalized)/ imputed data
- data_missing: discretized data/ missing entries left as NaN
- data_c_missing: continuous data/ missing entries left as NaN

Other outputs that could be used by other modules are:

- labels: a cell array of variable names
- mi: mutual information of the discretized variables with respect to a class variable
- studyid: ID of the imported patients
- FracSize: fraction size of the imported patients, required for NTCP calculation
- COMSI: superior-inferior location of a PTV centroid, required for NTCP calculation

It should be noted that the output data matrices (data, data_c, data_missing, data_c_missing) store each patient in one column and each variable in one row, which is a transpose of conventional row-based indexing, in order to be used as an input to the BNT toolbox. The last row of the matrices is reserved for a target class. The variable specified as a target can be modified in the file kyu_readphysical.m.

4 Variable Selection

Variable selection (filtering) is often a necessary step towards constructing a robust prediction model when the number of variables is large compared to the available examples. This package includes two filtering implementations: Koller-Sahami (KS) variable filtering and L1-regularized logistic regression (LASSO).

4.1 KS filter

This algorithm chooses a subset of variables in a semi-supervised fashion: every variable is measured, as its "usefulness", a class entropy with respect to a target class under the presence of other variables. The variables that already explains the target class is called the *Markov blanket*. The code implemented a backward elimination approach where it begins with a full set and iterates rounds of eliminations where one variable with the lowest cross entropy is removed from the set. Detailed theoretical explanation can be found in the original paper by Koller and Sahami [1]

The KS filter can be called in the following way:

```
>> [selected,CEmin,CEvar,CE_rand,blanket] = ...
    KSfilter(data,labels,N,k,verbose)
```

Here are the required arguments:

- data: discretized data with NaN imputed in ('data' output of kyu_BN_readdata.m)
- labels: variable names ('labels' output of kyu_BN_readdata.m)
- N: desired number of variables to be selected
- k: number of variables to include in the Markov blanket
- verbose: display messages (1) or not (0)

You can see the result of variable selection from the output argument "selected". For the information on other outputs, see a headnote in the file KSfilter.m.

The KSfilter.m requires the data dimensionality (N) and a blanket size (k) to be set by a user. For small datasets, larger k should be avoided in order to prevent over-fragmentation of data and zero counts from computation of conditional probability. For determining N, the original paper suggests observing the cross entropy of the removed variables as a function of elimination rounds. A sudden increase in the entropy can be taken as a good indication that the optimal dimensionality has reached. However, such a "kink" does not always appear. This implementation takes the approach of measuring the cross entropy of a pseudo variable filled with random values (CE_rand) and taking that value as a cutoff for the best dimensionality. Thus, the whole process of KS filtering is two sequential runs of Stability_KS.m, once to determine the number of variables and the second time to arrive at the final choice of variables. The function repeats the KS filtering in bootstrap replicates, which gives an option for uncertainty estimate on the results.

The following example shows how to use the function Stability_KS.m:

```
>> % first round of variable selection
>> k = 1; % blanket size 1
>> N = k+1; % remove the variables to the smallest set
>> Nrand = 1000; % bootstrap the KS 1000 times
>> verbose = 0; % don't display messages
>> [selected,CElist,CEvar_avg,CErand,blanket,labels]  = ...
        Stability_KS(k,N,Nrand,verbose);
The following variables are found:
1. a2m_pre
2. a2m_intra
3. a2m_end
...
choose the variables, separated by comma: 1,4,12,15
bootstrap sample #1
bootstrap sample #2
...
```

After completion of the first KS run, the results can be visualized for determining data dimensionality:

```
>> plotKSresult(CElist,CErand)
```

This will show a plot that looks like figure 2. Variables can safely be removed until its cross-entropy exceeds that of a random variable (indicating it gives no more than noise to a class distribution), which in this case leaves 6 out of 16 variables.

Then the second round of KS filtering is run:

```
>> % second round of variable selection
>> Nopt = 6;
>> [selected,CElist,CEvar_avg,CErand,blanket,labels]  = ...
        Stability_KS(k,Nopt,Nrand,0);
```

The first output argument "selected" is a binary matrix that stores selection results for every bootstrap runs (1: selected, 0: not selected). In order to see the frequency on which each variables are selected over the bootstrap

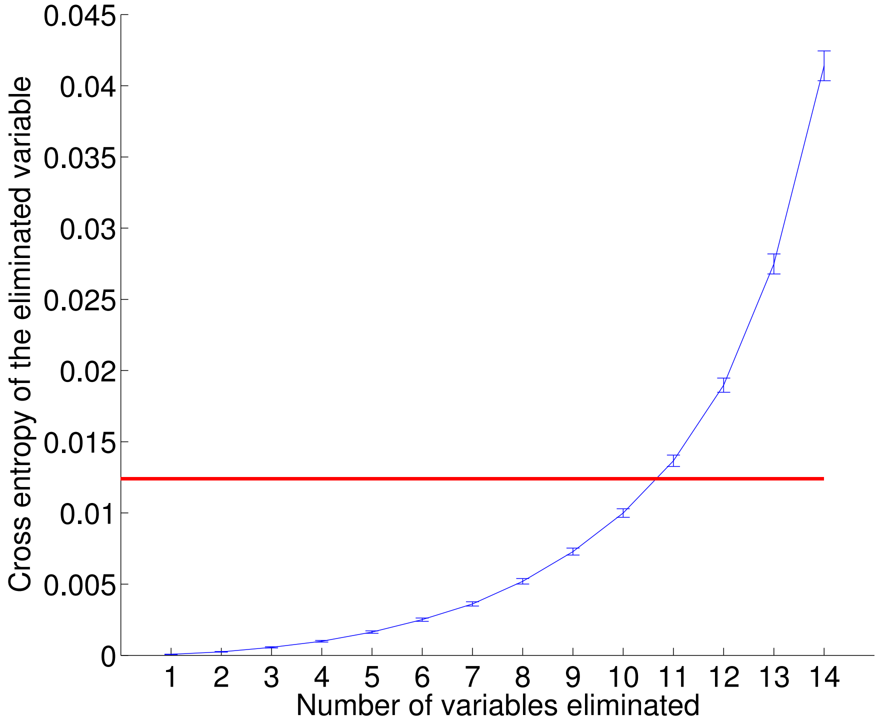

Figure 2: Cross entropy of the variables removed at each round of KS backward elimination. A red line indicates the cross entropy from a random variable.

runs, simply do:

```
>> selection_frequency = mean(selected,1);
```

4.2 LASSO

L1 regularization in logistic regression can induce sparsity in a solution by pushing some of the coefficient values towards zero. This filter fits the L1 logistic regression model to the data and selects the variables that have a non-zero coefficient. Similarly to KS, the selection is repeated in bootstrap for obtaining statistics. At each repetition, a shrinkage parameter (λ) of a L1 term is tuned to maximize the fit in a given bootstrap replicate, which is measured in two different metrics that users have to specify: mean square error (MSE) or area under the ROC curve (AUC).

```
>> Nrand = 1000; % bootstrap 1000 times
>> metric = 'AUC' % metric for tuning lambda: 'MSE' or 'AUC'
>> verbose = 0; % don't display messages
>> [selected,model_coal,lambda_hist,err,auc] = ...
    Stability_LASSO(Nrand,metric,verbose)
```

Same as the KS filter, you can look at the matrix 'selected' to see which variables are selected. 'lambda_hist' is a histogram of lambda values chosen during bootstrap repetitions. Note that the λ is chosen from a list of values that are hard-coded in Stability_LASSO.m. So users might need to do the first run, see the output 'lambda_hist' (histogram of lambda values chosen during bootstrap repetitions), and refine the search range of λ by modifying the variable 'lambda' in Stability_LASSO.m.

Inspired by a DREES functionality [2], The code also offers coalesced results which can be seen in a struct 'model_coal' in order to reduce the number of variable selection choices. Coalescence of two variable selection results occurs when the chosen variables are 'similar enough', which is measured by pairwise correlation. For example, if two selection differs by one variable, say a variable A in one set and B in the other, the two selections are considered the same if A is highly correlated with B.

5 Other preparatory steps

5.1 Bootstrap/CV data generation

After deciding which variables to carry to the modeling stage, the data will be trimmed to those variables. Unless you have a dedicated dataset for external validation, some examples within the original dataset has to be put aside for performance testing. The package generates such partition of data either in cross-validation (CV) or bootstrap settings. It takes the compressed data, creates several folds/replicates of partitions and saves them in a Matlab struct file.

```
>> val='BS'; % generate bootstrap replicates
>> Npart = 200; % bootstrap 1000 times
>> data = kyu_BN_GeneratePartition(Npart,val)
The following variables are found:
1. a2m_pre
2. a2m_intra
3. a2m_end
...
choose the variables, separated by comma: 1,4
```

The generates struct contains two fields: 'KM' and 'MI' indicating K-means and maximal mutual information methods were used for discretization, respectively. The two fields contain the same kinds of variables under them: the only differences are the subfields that are discretized. Subfield names are concatenated with strings that characterize what kind of information is stored in. Here is the list of the characterizing strings:

- *train*: values for training instances
- *test*: values for testing instances
- *missing*: missing data left as NaN
- *nointra*: intra-treatment biomarker data from *missing* is erased and replaced with NaN.
  Subfields without *missing* or *nointra* are imputed with a method as specified in the parent field name (KM or MI).
- *c*: continuous data (discretized otherwise)

- *bio*: data that consists of biological variables and a class
- phy: prediction values from custom physical models (e.g. TCP, NTCP) for the testing instances (nothing is generated for training instances). Currently zeroed out, left to a user to create one.
- *orig*: the source data with the original dimension (number of variables X sample size)
- *patient*: Instead of variable values, patient IDs are stored for each partition. Note that this ID is not the same as the studyID from the kyu_BN_readdata.m: the numbers simply point to a column number of the original data matrix.

5.2 Defining a graph prior

A graph prior is defined as a set of links between variables with their weights indicating prior likelihood of that relationship. A graph prior is integrated into MCMC sampling which as a result yields the Bayesian solution of a causal graph: an ensemble of graphs and their posterior probability.

There are two modes of a graph prior:

- Causal prior: Non-causal links are given a weight of 0, and a equal weight of 0.5 is assigned to all the other links. Currently, non-causality is set between categories of variables in a code make_dagprior.m. An example of causality relationships is shown in figure 3. Variables are categorized into one of the 5 groups at kyu_BN_RP_CategorizeVariables.m which is called within make_dagprior.m. The graphs that contain non-causal links will not be sampled during the MCMC routine and therefore excluded from an ensemble of graphs.
- Biological prior: It is built upon the causal prior with the causal links assigned different weights depending on the level of confidence/ prior knowledge on the correlations. This concept is based on the work by Werhli and Husmeier [3]. In the current implementation, there are 3 levels of confidence originally designated for direct, indirect, and unknown protein-protein interactions.

To create a prior graph, load the data and choose the variables using kyu_BN_readdata.m and then run make_dagprior.m. It takes as inputs

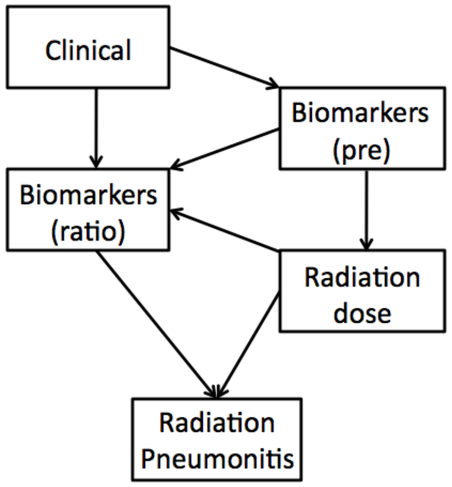

Figure 3: Diagram of allowed causal links between variable categories used for accepting/rejecting graph samples during MCMC simulation.

a list of variable names (labels) and a prior type and generates a matrix representation of a graph prior (dag_prior) which has a dimension of Nnodes*Nnodes (Nnodes: the number of variables) and each (i,j)-th element representing a prior weight for the relationship (i-th variable $\rightarrow$ j-th variable).

```
>>  [data,data_c,data_missing,data_c_missing,...
    labels,mi,studyid,FracSize,COMSI] = ...
    kyu_BN_readdata(SBRTfilter,disc)
...
>> prior type = 1; % 1: prior mode. 1:causality, 2:biological
>> [dag_prior,mask_caus,mask_ipa,category] = ...
    make_dagprior(labels,prior_type)
```

6 MCMC graph learning

The MCMC module extends from a Bayesian Network graph learning part from the BNT toolbox. This toolbox modified the Metropolis-Hastings graph sampling subroutine from the BNT to accommodate the graph prior. Currently by default the algorithm (especially a graph-scoring part) assumes fully observed and binarized data and thus cannot handle NaN or continuous data. If more than 2 bins is to be used, a hyperparameter args_graph.ns needs to be modified accordingly.

Causal and biological prior are handled differently in the modified sampling algorithm (can be seen at learn_struct_mcmc_L1.m):

- Causal prior: At each step of a random walk, a proposal density is modified so that it blocks the moves from the current graph that creates any of the non-causal links.
- Biological prior: When computing an acceptance ratio, A Bayes factor is multiplied by a prior factor (pf) which gives a value > 1 if the move improves the agreement with a prior. In addition, a random walk in a graph space is followed by a walk in the prior hyperparameter β. The implementation follows the methodology by Werhli & Husmeier [3].

The major computation parts of the codes are parallelized and requires a cluster profile to be set up in the MATLAB R2013 and above. By default, one worker (core) is assigned to each MCMC chain from a single initialization.

For brief introduction, consult a BNT toolbox code manual (`http://bnt.googlecode.com/svn/trunk/docs/usage.html`) or Geyer [4].

6.1 MCMC chain length estimation

Prior to a full bootstrapped graph learning, a sample run with an original training data is recommended to estimate the speed of convergence. First, create the bootstrapped data (section 5.1) and call the function kyu_BN_MCMC_convtest:

```
>>  data = kyu_BN_GeneratePartition(Npart,val)
...
>> chainlength = 50000; % MCMC chain length
>> whichprior = 'causal'; % causality prior
>> cluster = 'guillimin'; % name of the cluster
>> disc = 'KM'; % data discretization method to apply. 'KM' or 'MI'
>> [R,params] = ...
     kyu_BN_MCMC_convtest(data,chainlength,whichprior,cluster,disc)
```

Except for a prior type (causal/biological) and a MCMC chain length, all the other simulation hyper parameters need to be set inside the file kyu_BN_MCMC_convtest.m, prior to the execution. The used parameter values can be reviewed retrospectively in an output argument 'params'. The graph related hyper parameters (params.graph) is filled in automatically and not tunable. However, the following MCMC-related parameters (params.MCMC) can be set by a user :

- burnin: burn in period (the number of first samples to discard)
- ScoringFn: a graph scoring function. 'bayesian' or 'bic' (Bayesian Information Criteria).
- alpha_d: equivalent sample size for a Dirichlet parameter prior. Larger value induces more links but may affect the convergence.
- NoInit: number of random initialization graphs to generate. Higher NoInit will improve mixing. This will create parallel jobs of MCMC, one job per a chain initialized by different random graphs.
- InitDensity: a degree of sparsity in initial graphs. Lower value leads to initialization with fewer links.
- InitBeta: (biological prior only) a starting value for a strength parameter β for a biological prior. (see [3]). β is also sampled at the same time as graphs during MCMC, which leads to almost twice longer simulation time.
- NoTopGraphs: the number of highest posterior graphs to keep in an ensemble (to save memory. small posterior graphs contribute little to Bayesian prediction). It is not enforced for convergence testing (i.e. all the graphs are kept).
- thinning: downsampling ratio applied to samples. Fewer number of samples are kept to save memory without significantly affecting graph posterior.
- MaxParents: the maximum number of parents a node can have. It has to be kept small for small sample size to prevent zero counts when conditional probability is estimated from data.

After the number of iterations specified by ChainLength has passed, various statistics about the samples will be obtained and saved into an output argument "R". Some statistics are measured throughout the chain at the frequency specified by "thinning" (denoted as t). Other metrics concerning posterior distribution of samples are measured at larger frequency specified by "gsfreq" (denoted as T). At this frequency, the graphs collected up to T are histogrammed to create a posterior distribution from which other statistics are derived.

1. evaluated every t:
    - ACR: acceptance rate
    - Beta_track: (biological prior only) sampled prior strength hyperparameter
    - agree_caus: degree of conformity of sampled graphs to causality prior. With the current version of Metropolis-Hastings sampler, the causality is always enforced and it should stay at 1.
    - agree_ipa: (biological prior only) degree of conformity of sampled graphs to biological prior.
    - score: Average score of graph samples across the N=NoInit chains. A scoring function of choice (args_MCMC.ScoringFn) is applied.
    - score_upper: upper bound of the score (top 10% quantile)
    - score_lower: lower bound of the score (bottom 10% quantile)
2. evaluated every T:
    - DAGhistogram: posterior distribution of sampled graphs. It consists of two subfields: 'matrix' for a graph matrix and 'frequency' for its posterior probability at a chain length T.
    - TopGraph: Maximum a-posteriori (MAP) Bayesian Network graph.
    - histpeaked: full-width half maximum of the posterior distribution.

Currently, there is no implementation of a quantitative method to determine acceptable convergence or mixing. Instead, progress of MCMC sampling can be visualized using the function kyu_PlotResultsfromMCMC:

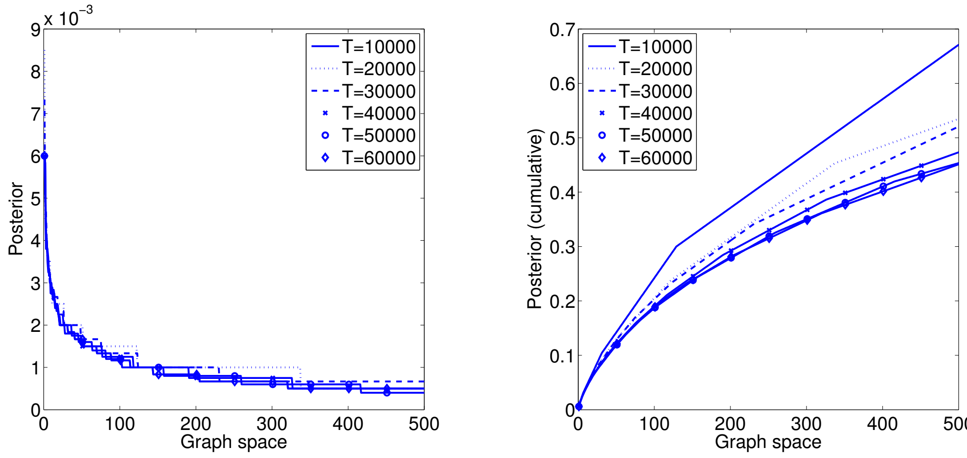

Figure 4: Posterior estimation of graphs over MCMC runs up to 60000 iterations (T). Left: differential, right: cumulative.

```
>> [R,params] = ...
     kyu_BN_MCMC_convtest(data,chainlength,whichprior,cluster)
>> % specify the iteration numbers to show in a plot
>> timept = 10000:10000:60000;
>> kyu_PlotResultsfromMCMC(R,timept)
```

The displayed information includes changes in posterior distribution, both differential and cumulative (figure 4) as well as acceptance ratio and an average graph score (figure 5)

6.2 Validation of MCMC graph learning

After the graph learning hyper parameters are tuned and convergence was checked, the graph learning process will be repeated in bootstrap/cross validation partitions should internal model validation be done. Open the code kyu_BN_GraphLearning.m and put in the optimized hyperparameters as they were previously set in the code kyu_BN_MCMC_convtest.m.

```
>> % first need to create partitions
>>  data = kyu_BN_GeneratePartition(Npart,val)
...
>> chainlength = 60000; % long enough chain length for convergence
>> whichprior = 'causal'; % which type of prior
```

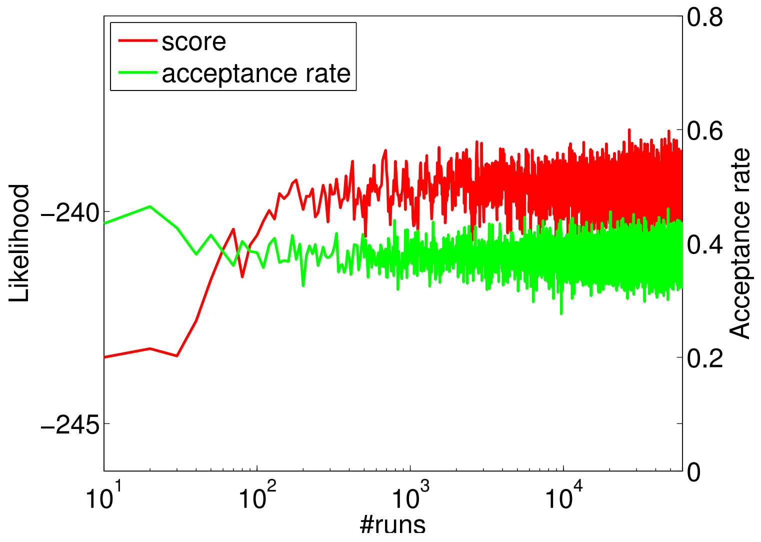

Figure 5: Change in likelihood graph score and acceptance rate over 60000 MCMC iterations shown on log scale. 25 Chains with random initialization were averaged.

```
>> clustername = 'guillimin'; % name of the cluster
>> validation = 'bootstrap'; % 'bootstrap', 'cv', or 'fitting'
>> disc = 'KM'; % data discretization method to apply
>> res_previous = []; % set empty if first time run
>> index = 1:10; % create a list of bootstrap/CV indices to use.
                 % first-time run, set 1:(number of replicates)
                 % if some jobs crash, the learning can be re-run for
                 % the sets specified in this variable
                 % should be set as 1 when validation = 'fitting'
>> result = kyu_BN_GraphLearning(data,ChainLength, ...
                whichprior,clustername,validation,disc, ...
                index,res_previous);
```

When the run is complete, the code will return a cell array 'result' with each element storing the learning results (explained later). If any of the parallel jobs crashes (due to troubles in a cluster for example), a message will be shown containing the partition indices where the crash happened:

```
set 3 4 crashed
set 7 8 crashed
```

As well, the elements in the cell 'result' corresponding to the crashed partition indices will be left empty. In that cases, the crashed indices are collected and used for the second run of kyu_BN_GraphLearning_submit.m. Note that the cell array 'result' from the previous run should be provided as an input:

```
crashed = [3 4 7 8];
>> result = kyu_BN_GraphLearning_submit(data,ChainLength, ...
                whichprior,clustername,validation,disc,crashed,result);
```

(warning: a variable FoldsPerJob in kyu_BN_GraphLearning.m shouldn't be modified at the re-run!)

The completed result is converted into a single struct variable that stores the learned graphs from all the bootstrap/CV replicates with a following command:

```
>> output = kyu_BN_GraphLearning_collect(result);
```

The final result 'output' consists of three fields:

- gs_top: a matrix storing the highest posterior DAGs. It has a dimension (number of replicates X args_MCMC.NoTopGraphs). Each row stores the graphs sampled from a single bootstrap/CV replicate. Note that NoTopGraphs should be set high enough because low posterior graphs beyond this number will not be saved and cannot be retrieved.
- posterior: a matrix of posterior probability corresponding to graphs in the gs_top.
- beta_hist: (biological prior only) stores the distribution of sampled β (prior strength).

7 Parameter learning and class inference

Parameters of a Bayesian Network refer to conditional probability values for every parent-child relationships in a DAG. This package borrows the expectation-maximization (EM) version of maximum likelihood parameter learning from the BNT, which is able to handle missing data.

In this package, parameter learning and prediction is made sequentially in the same code kyu_BN_BMA.m. The code first takes as an input a training set and high posterior graphs acquired from the MCMC module using the same training set. Then it EM-learns parameters for every graphs in an ensemble (kyu_BN_paramlearn.m), which in turn creates a Bayesian Network and junction tree objects ('bnet' and 'engine' from kyu_BN_paramlearn.m, respectively). When a testing example is presented, the two objects are used for probabilistic inference on a class variable or, in other words, obtaining $P_i(class)$ for the i-th graph in an ensemble. This is performed in a code kyu_BN_ClassPs_Bayesian.m.

The inference results from individual graphs are combined to obtain an ensemble solution of $P(class)$ which is mathematically defined as the following:

$$P(class) = \frac{\sum_{i=1}^{N} P_i(class) P(i|D)}{\sum_{i=1}^{N} P(i|D)} \tag{1}$$

where $P(i|D)$ indicates a posterior probability of a graph i estimated by MCMC. This process takes place at a code kyu_BN_perftest_Bayesian.m.

One additional feature of this module is evaluation of polarity of relationships found in a Bayesian Network. A polarity of a value 1 is given between a parent p and a child c if $P(c = 2|p = 2) > P(c = 2|p = 1)$ (positive correlation) and -1 for the opposite case. However, the output 'polarity' returned from kyu_BN_BMA.m is an average value of the polarity across an ensemble. The average is obtained for different ensemble sizes as given by an input argument EnsembleSizes.

A wrapper function kyu_BN_ParamLearnPredict_submit.m gives users an option to set the necessary parameters and an option for distributed computation. Here is an example of calling this function:

```
>> %create partitions
>>  data = kyu_BN_GeneratePartition(Npart,val);
...
>> clustername = '...';
>> validation = 'bootstrap';
>> whichprior = 'causal';
>> disc = 'KM';
>> % learn the graphs
>> MCMCresult = kyu_BN_GraphLearning(data,ChainLength, ...
                whichprior,clustername,validation,disc,index,[]);
>> % prediction/polarity will be evaluated as a function of
>> % different ensemble sizes given to this array
>> EnsembleSizes = [1 5 10 50];
>> % 'nointra' drops the variables acquired at later time pts when making
>> % probability inference. This option reduces the information used for
>> % inference in order to test the reasoning ability of the graph ensemble
>> % if not desired, choose 'missing' (this will still probabilistically
>> % handle the missing entries originally present in the raw dataset
>> whichinput = 'nointra';
>> % similarly to the graph learning module,
>> % there is an option to re-run crashed jobs
>> % below is the setup for the first time run
>> res_previous = []; % set empty if first time run
>> index = 1:Npart; % create a list of bootstrap/CV indice to use.
>> res = kyu_BN_ParamLearnPredict_submit(...
data,MCMCresult,whichinput,clustername,validation,disc, ...
EnsembleSizes,index,res_previous);
```

After the calculation is complete, an extra step is required to combine the results from workers.

```
>> res = kyu_BN_ParamLearnPredict_submit(...
data,MCMCresult,whichinput,clustername,validation,disc, ...
EnsembleSizes,index,res_previous);
>> [Probs,polarity] = kyu_BN_ParamLearnPredict_collect(data,res)
```

The first output 'Probs' represents the Bayesian averaged P(class) for every instances in training/testing sets. Results from different ensemble sizes are divided into columns. The second output 'polarity' returns the polarity matrix averaged over folds as well as varying sizes and ensemble.

8 Classification Performance

The obtained probability values can now be used for classification performance tests such as receiver operating characteristics (ROC). The package includes an implementation of 0.632+ bootstrap ROC metrics [6] (kyu_Perf_632BSplus.m). For this type of validation, it is required to build another ensemble model with the entire dataset as a single training set and generate the class probability estimates from it. This can be done by setting a input parameter 'validation' to 'fitting' when calling kyu_BN_GraphLearning_submit.m and kyu_BN_ParamLearnPredict_submit.m.

Below is an example usage:

```
>> % Probs: P(class) from bootstrap
>> % Probs_fitting: P(class) trained from the whole training data
>> % data: partitioned data struct
>> [Perfs_632p,Perfs_632pM] = kyu_Perf_632BSplus(Probs,Probs_fitting,data)
```

The cell Perfs_632p contains the means and standard errors of the following classification performance estimated in 632+ bootstrapping:

- .auc: area under the ROC curve.
- .se_opt: sensitivity at an optimal operating point
- .sp_opt: specificity at an optimal operating point
- .ROCY: Y-axis points (true positive rate) of a bootstrap-averaged ROC curve. X-axis points (false positive rates) are fixed at regular intervals (0:0.01:1)

The cell Perfs_632pM contains the raw samples of performance metrics from each bootstrap folds and could be used for statistical testings (e.g. model comparison).

References

[1] D. Koller and M. Sahami, "Toward optimal feature selection," technical report, Stanford InfoLab, 1996.

[2] J. D. Bradley, A. Hope, I. El Naqa, A. Apte, P. E. Lindsay, W. Bosch, J. Matthews, W. Sause, M. V. Graham, and J. O. Deasy, "A nomogram to predict radiation pneumonitis, derived from a combined analysis of RTOG 9311 and institutional data," *International journal of radiation oncology, biology, physics*, vol. 69, no. 4, pp. 985–992, 2007.

[3] A. Werhli and D. Husmeier, "Reconstructing Gene Regulatory Networks with Bayesian Networks by Combining Expression Data with Multiple Sources of Prior Knowledge", *Statistical Applications in Genetics and Molecular Biology*, vol. 6, no. 1, pp. 1–47, 2007.

[4] C. Geyer, "Introduction to Markov Chain Monte Carlo", in *Handbook of Markov Chain Monte Carlo*, pp. 3-48, Taylor & Francis, 2011.

[5] N. Friedman, M. Goldszmidt, and A. Wyner, “Data analysis with bayesian networks: A bootstrap approach,” in *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI’99, (San Francisco, CA, USA), pp. 196–205, Morgan Kaufmann Publishers Inc., 1999.

[6] B. Efron and R. Tibshirani, “Improvements on cross-validation: The .632+ bootstrap method,” *Journal of the American Statistical Association*, vol. 92, no. 438, pp. 548–560, 1997.