Implementing conditional Gaussian scoring function in pygobnilp and learning Bayesian network from complete mixed data

Yasuaki Nakayama
dept. Computer Science
University of York
York, UK
yn659@york.ac.uk

***Abstract*—Nowadays, Bayesian networks are widely used for various applications, such as medical diagnosis systems, which have a sort of uncertainty. Because of its broad usability, many scientists and engineers have studied methods to automatically learn Bayesian network structures from data. For example, pygobnilp is a program developed by Cussens that supports learning a Bayesian network from complete data. However, pygobnilp was not able to deal with mixed data. Therefore, in this project, a method that can evaluate a Bayesian network from a mixed dataset was implemented in pygobnilp, and this made it possible to learn every kind of Bayesian networks. The evaluation was achieved by the conditional Gaussian score, which was proposed by Andrews et al. A series of experiments revealed that the implemented method correctly scored a Bayesian network, and the efficiency of the optimized method was higher than bnlearn, which consists of R packages for learning Bayesian networks.**

***Index Terms*—Bayesian networks, log-likelihood, conditional Gaussian score, mixed data, bnlearn, pygobnilp**

I. Introduction

A *Bayesian network (BN)* is one of the probabilistic graphical models which represents probabilistic relationships among a finite number of variables. Modeling all the conditions and exceptions with which one must deal when modeling a real-world application is known to be impossible. The uncertainty derived from this fact makes it difficult to implement an intelligent system for the application. However, as the probabilistic relationships encoded in a BN enable efficient inference on the random variables under several conditions, BNs are widely employed to conduct inference with uncertainty.

For example, Wiegerinck et al. [1] established several inference applications by using BNs: e.g. a medical diagnosis decision support system; a petrophysical decision support system; and a victim identification system. First, their system for medical care was intended to assist physicians in diagnosing outpatients and to educate medical students. The second application was designed to predict compositional volume fractions in a reservoir according to measured data and the effect of additional measurements. Finally, the system for identifying victims handled DNA profiles and pedigree information as well as a large number of events that caused hundreds of victims and missing persons, which enabled the correct kinship analysis. In these ways, many studies have been conducted to employ BNs as an inference model to perform effective reasoning.

Briefly speaking, a BN structure consists of two components: a *directed acyclic graph (DAG)* and a set of parameters. A DAG illustrates the relationships over the random variables in a BN. For example, a DAG $A \rightarrow B$ represents that the variable B is probabilistically determined by the variable A. These probabilities among variables are stored in the set of parameters. Several researchers have studied how to efficiently learn a DAG which express the plausible relationship given a practical dataset.

One of the most popular methods to learn a structure of BN is based on *Integer Programming (IP)*. This method aims to find a BN structure that maximizes a scoring function, such as BDeu [2]. This function should be a linear function calculated from a supplied data and be able to be decomposed into local scores for each variable. Since scoring functions measure the appropriateness of a BN structure, the method can search for the most probable BN given a dataset.

Now, several programs for learning and dealing with BNs are available as open-source software, such as bnlearn [3] and tetrad [4]. In particular, this paper will focus on pygobnilp which was implemented by Cussens [5]. This is a Python program for learning BNs from complete data and is derived from GOBNILP, or Globally Optimal Bayesian Network learning using Integer Linear Programming. GOBNILP is originally a C program and provides many functions that are useful for learning BNs based on integer programming and local scores. Similarly, the library of pygobnilp can assist in developing a learning system of BNs from data.

Although the current pygobnilp can produce plausible BNs from either discrete or continuous data, it cannot handle a dataset that consists of both discrete and continuous values. As both kinds of variables are often involved in many applications, a program should be able to learn from mixed data. For example, the possibility of a certain disease will be determined by many types of factors, while these factors could be discrete values (e.g. whether the patient smokes) as well as continuous values (e.g. the blood pressure). Therefore, the purpose of this

project is to implement methods in pygobnilp that can deal with mixed data and that can establish a BN with both discrete and continuous variables.

The remainder of this paper is structured as follows. First, several prior studies related to learning a BN from data are reviewed in Section II. Afterward, an overview of BNs and structure learning is illustrated in Section III. Next and foremost, the scoring function that is employed as a method of pygobnilp is described in Section IV. Section V provides the results of a series of experiments and discussion on them, and finally, this paper concludes in Section VI.

II. Related Work

Several researchers have proposed various BN structure learning methods from a complete dataset. An effective approach was suggested by Cussens, in which a BN structure can be learned from a discrete dataset [6]. He adopted integer programming as a search algorithm and *log marginal likelihood*, or the *BDeu* score, as a scoring function. In his approach, the constraints that every BN structure must satisfy was rationally implemented, and specific user constraints can also be defined.

Later, Barlett and Cussens reinforced this method so that it can more efficiently learn a BN [2]. They implemented an approach that accelerates the search for cutting planes which are significant for integer programming, as well as a fast greedy algorithm that finds higher-scoring BNs. Furthermore, their investigation into the convex hull of DAGs suggested its usefulness for learning BN structure.

Although the research discussed above used only the BDeu score to evaluate a DAG of a BN, many scoring functions have already been proposed. Carvalho evaluated some *information-theoretic* scores (namely LL, AIC, BIC, MDL, and MIT) as well as *Bayesian* scores (namely K2, BD, BDe, and BDeu) [7], all of which can be a scoring function of a BN structure. Employing a classification task for this evaluation, he revealed that information-theoretic scores performed better than Bayesian scores in many cases.

While all of these studies focused on discrete data, others aimed to learn a BN from continuous data. An empirical study in learning BNs with continuous variables was conducted by Qian and Miltner [8]. In their proposed methods, the inference was achieved by a regression model, and therefore, the most probable BNs were selected according to the error variances between deduced values and actual ones. They evaluated their method by using continuous data for building nutrient standards in streams and rivers in Ohio, US.

Many studies, including these works concisely illustrated above, assumed that supplied data is either discrete or continuous, but they do not accept a dataset that includes both kinds of values. Since both types of variables are involved in many practical applications, it is highly significant to implement a method that can learn BNs from mixed data. However, there have been very few studies that proposed an effective approach to realize this.

For example, Liu et al. briefly described a discretization technique, the *minimum description length (MDL)*, and empirically evaluated several scoring functions [9]. As discretization enabled the learning algorithm to deal with a continuous variable as a discrete one, they were able to learn BNs from mixed data. Furthermore, Chen et al. suggested a discretization method that can find an effective policy to segment continuous values into several groups [10]. This research reinforced the learning algorithm, and they showed its superiority to the conventional method.

As was shown in these studies, the discretization of continuous values was effective for learning BNs to some extent. However, this may result in low credibility of a learned BN because discretizing continuous values possibly loses significant information in the supplied data. Therefore, a method that encourages learning from mixed data without discretization is required to effectively adapt BNs to a wide range of applications.

A framework that would satisfy these requirements was proposed by Andrews et al [11]. They provided a general explanation of how the learned BNs can perform inference without discretization, and then two scoring functions, the *Conditional Gaussian (CG)* score and the *Mixed Variable Polynomial (MVP)* score, were proposed to evaluate each possible BN. Since they found that one of their suggestions, CG, resulted in high performance, the goal of this project is to implement their approach in pygobnilp.

III. Bayesian Networks

This section provides an overview of a BN and its learning procedure. Firstly, the general theory of BNs is concisely described, and next, an approach to exactly learning the most plausible BN structures from complete data is briefly explained.

A. General theory of Bayesian networks

First and foremost, a BN is represented as a *directed acyclic graph (DAG)* $G = (V, E)$ which is a directed graph with the absence of cycles. A DAG has n random variables $V = \{X_1, X_2, \ldots, X_n\}$ and a set of arrows E that represent direct dependencies between variables. Each variable takes either discrete or continuous value, and an arrow $(X_i, X_j) \in E$ represents that the variable X_j is directly influenced by the variable X_i. In this case, the variable X_i and X_j are called a *parent* of X_j and a *child* of X_i, respectively.

For example, Figure 1 elucidates the DAG of a BN, called 'Asia', which was introduced by Lauritzen and Spiegelhalter [12] and briefly summarized by Cussens [6]. This represents a probabilistic model for a medical 'expert system'. Each variable can be either TRUE (t) or FALSE (f), and each of them means as follows: A = "visit to Asia", T = "Tuberculosis", X = "Normal X-Ray result", E = "Either tuberculosis or lung cancer", L = "Lung cancer", D = "Dyspnea (shortness of breath)", S = "Smoker", and B = "Bronchitis". It is evident from this BN that variable D is a child of parents E and B, which visually tells that bronchitis and either tuberculosis or

lung cancer directly influence the probability to cause dyspnea. In this way, BNs efficiently present probabilistic relationships.

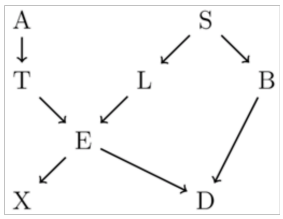

Fig. 1. A DAG of a BN with 8 random variables A, T, X, E, L, D, S, and B (from [6], page 100).

Besides a DAG, a BN holds a set of parameters $\boldsymbol{\theta}$, which consists of conditional distributions for each BN variable. For example, parameters of variable D in the BN shown in Figure 1 can be represented as a *conditional probability table (CPT)*, which specifies a distribution for the variable for each possible joint instantiation of its parents. Particularly, the CPT for variable D could be:

$$\begin{cases} P\left(D=t|B=f,E=f\right)=0.2 \\ P\left(D=f|B=f,E=f\right)=0.8 \\ P\left(D=t|B=f,E=t\right)=0.6 \\ P\left(D=f|B=f,E=t\right)=0.4 \\ P\left(D=t|B=t,E=f\right)=0.1 \\ P\left(D=f|B=t,E=f\right)=0.9 \\ P\left(D=t|B=t,E=t\right)=1.0 \\ P\left(D=f|B=t,E=t\right)=0.0 \end{cases} \tag{1}$$

If a BN holds continuous variables, parameters could be represented as coefficients of functions, which are estimated by a linear regression model.

By multiplying the relevant conditional probability distributions defined by the parameters, the joint probability distributions of any sets of variables can be calculated efficiently. For example, the expert system displayed in Figure 1 has 2^8 joint instantiations of variables, but the number of parameters of the BN is much fewer than that, and thus, the representation of the BN would be much smaller. This is enabled by the capability of BNs to encode *conditional independence* of the random variables. In short, two variables X_i and X_j are conditionally independent given a variable X_k with $P(X_k) > 0$ if $P(X_i \cap X_j) = P(X_i|X_k)P(X_j|X_k)$ [13]. In the BN of Figure 1, as variable E is dependent on A via T, A and E are conditionally independent given variable T. In this case, without the information on variable T, the value of A affects the probability of E, but once the data on T is provided, variable E is not influenced by A anymore. This is because the definition of conditional independence leads to the probabilistic relationship $P(X_i|X_j, X_k) = P(X_i|X_k)$.

This representation of a system enables more effective inference on random variables. Domain experts may be able to build a BN for a system, but such a manual approach has several problems; for example, experts' time is limited, experts may disagree, and they may make mistakes. Thus, many computer scientists have sought an approach to reliably and efficiently obtain a BN from data. In the following section, one such method which is adopted in pygobnilp is described.

B. Integer programming

To learn exactly a BN structure from data, integer programming (IP) is employed in pygobnilp. Generally, IP aims to find the solution that maximizes a linear objective function of the given problem. It is necessary to consider the constraints that a solution must satisfy as well as the constraint that every variable must take an integer value. Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ be the problem variables where each variable must take an integer value ($x_i \in \mathbb{Z}$). In this case, the linear objective function of the IP problem can be represented as $\boldsymbol{c}^T\boldsymbol{x}$, where $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ is a real-valued vector ($c_i \in \mathbb{R}$) of objective coefficients for each problem variable. Next, linear constraints are introduced, which are of the form $A\boldsymbol{x} \leq \boldsymbol{b}$ where $\boldsymbol{b} = (b_1, b_2, \ldots, b_m)$ is the real-valued vector ($b_i \in \mathbb{R}$) and A is the real-valued matrix whose shape is $(m \times n)$. The goal of an IP problem is, therefore, to find the solution $\boldsymbol{x}$ that maximizes the linear objective function $\boldsymbol{c}^T\boldsymbol{x}$ and that satisfies the series of constraints $A\boldsymbol{x} \leq b$.

In order to solve an IP problem, pygobnilp uses the *linear relaxation* of the original problem. A linear relaxation is almost the same as the original IP problem, but it allows the variables to take non-integer values. By introducing the linear relaxation and additional two steps: *cutting planes* and *branching* [6], the supplied IP problem is solved efficiently. A cutting plane is a linear inequality that rules out the solution to the linear relaxation, and it recreates a new linear relaxation that has a stricter upper bound. This raises the possibility that a solution to the linear relaxation is entirely integer-valued.

In branching, an IP variable x_i and an appropriate integer value l are selected together, where the linear relaxation solution value of x_i is a non-integer, and then, two new sub-problems are created: one has a new constraint $x_i \leq l - 1$; and the other has a new constraint $x_i \geq l$. If the upper bound on the best solution for a sub-problem is below the best solution found so far, the optimal solution for the sub-problem is concluded to be worse than the best solution, and thus, the sub-problem can be disposed of. By implementing these techniques, pygobnilp solves the IP problem for learning BN structure.

C. Modeling Bayesian network learning problem as an IP problem

It is known that many problems including learning structures of a BN can be encoded into an IP problem [6]. This section illustrates how the BN structure learning problem is encoded into an IP problem. First, variables for this IP problem are defined as binary 'family' variables $I(W \to v)$ for each variable v in a BN and its candidate parent set W, where $I(W \to v) = 1$ iff W is the parent set of v and $I(W \to v) = 0$ otherwise. For example, the DAG in Figure 1 would be represented by a solution where $I(\emptyset \to A) = 1$,

$I(\emptyset \rightarrow S) = 1$, $I(\{A\} \rightarrow T) = 1$, $I(\{S\} \rightarrow L) = 1$, $I(\{S\} \rightarrow B) = 1$, $I(\{L, T\} \rightarrow E) = 1$, $I(\{E\} \rightarrow X) = 1$, $I(\{B, E\} \rightarrow D) = 1$, and all other IP variables have the value 0.

Next, to find out the 'most probable' DAG, the linear objective function, which evaluates a candidate BN structure given a dataset, must be defined. Given data $Data$, a DAG G is evaluated by $P(G|Data)$, which is the posterior probability of the DAG taking the data into account. Here, the probability of a BN G given the data $Data$ is denoted as $P(G|Data)$. According to Bayes' theorem, this can be rewritten as:

$$P(G|Data) = \frac{P(Data|G)P(G)}{P(Data)}. \tag{2}$$

In this equation, $P(G)$ is the prior probability of the DAG G, which represents the state of knowledge before seeing the data. $P(Data|G)$ is called *marginal likelihood* or *Bayesian evidence* [14], which encodes how the degree of plausibility of the DAG G changes when new data is given. This marginal likelihood marginalizes the BN parameters $\boldsymbol{\theta}$ and is defined as:

$$P(Data|G) \equiv \int_{\Omega_G} P(Data|\boldsymbol{\theta}, G)P(\boldsymbol{\theta}|G)\mathrm{d}\boldsymbol{\theta} \tag{3}$$

where Ω_G is the parameter space of the BN defined by G. $P(Data)$ remains constant across different structures of BNs because it only depends on the supplied data. This leads to the relationship $P(G|Data) \propto P(Data|G)P(G)$. If the prior bias between candidate BNs does not exist, the prior probability $P(G)$ is assumed to be stable for all G. Thus, in this case, the BN that maximizes the marginal likelihood $P(Data|G)$ or log marginal likelihood $\log P(Data|G)$ would be the most probable.

The log marginal likelihood can be expressed as a linear function of the family variables $I(W \rightarrow v)$. It can be decomposed into 'local scores' $s(v, W)$, which is computed from data for each family. The IP for learning BN structure would be the problem of maximizing the global score $S(G|Data)$, which can be expressed as:

$$S(G|Data) = \sum_{v,W} s(v, W)I(W \rightarrow v) \tag{4}$$

The solution $I(W \rightarrow v)$ that maximizes $S(G|Data)$ would be the most probable DAG for the given data.

Based on the definition of a DAG, the following two constraints must be added. First, each BN variable must have exactly one set of parents (including the empty set). Therefore, letting V be the set of BN variables, this constraint can be encoded as:

$$\forall v \in V : \sum_{W \subseteq V} I(W \rightarrow v) = 1 \tag{5}$$

where W is a potential parent set of the variable v.

Second, a DAG must not have any cycles, which, by using a cluster, Jaakkola [15] suggested to express as:

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : W \cap C = \emptyset} I(W \rightarrow v) \geq 1 \tag{6}$$

where cluster C is a subset of BN nodes. A DAG does not have any cycles if it satisfies this constraint, and this can be proved by considering its contrapositive. Suppose a directed graph $G = (V, E)$ which has cycles. If a variable set $C = \{v_1, v_2, \ldots, v_m\}$ ($C \subseteq V$) forms a cycle, $\forall v_i \in C : I(W \rightarrow v_i) = 0$ for every variable set W such that $W \cap C = \emptyset$ because every variable in C has at least one incoming arrow from a node in C. As a simple example, consider the graph shown in Figure 2 that has a cycle $C = \{v_1, v_2, v_3, v_4\}$. The IP variables of the nodes in the cycle are $I(\{v_4, v_7\} \rightarrow v_1) = 1$, $I(\{v_1\} \rightarrow v_2) = 1$, $I(\{v_2\} \rightarrow v_3) = 1$, $I(\{v_3\} \rightarrow v_4) = 1$, and otherwise 0. Therefore, if a graph $G = (V, E)$ has cycles, a cluster $C \subseteq V$ must exists such that $\sum_{v \in C} \sum_{W : W \cap C = \emptyset} I(W \rightarrow v) < 1$. Thus, if the constraint illustrated above is true, the graph must not have any cycles.

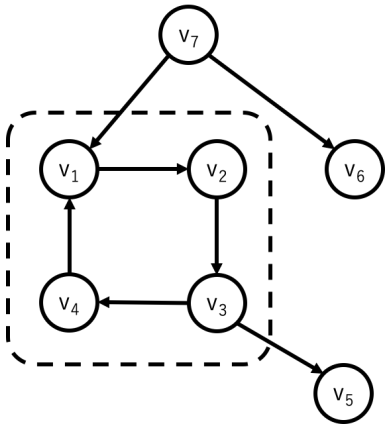

Fig. 2. A directed graph with nodes $v_1, v_2, \ldots, v_7$. Let the set of variables in the square be C.

These two constraints are implemented in pygobnilp to satisfy the definition of a BN: a variable must have one parent set; and its DAG must not have any cycles. As the latter generally requires too many cluster constraints to add, however, pygobnilp starts with few of these and dynamically add others as cutting planes. Each time the linear relaxation of the IP problem is solved, a cluster constraint, which the linear relaxation solution does not satisfy, is searched. If no cluster constraints can be found and the solution to the linear relaxation is integer-valued, the solution would be the solution to the original IP problem. If no cluster constraints can be found and the solution to the linear relaxation is fractional, new cutting planes, which are not 'cluster' constraints, are searched.

Figure 3 summarizes the procedures to solve an IP problem. Each time the linear relaxation is solved, cutting planes are searched and, if some are found, they are added to the problem. If no cutting planes are found, two new sub-problems are created by branching. To solve the linear relaxation of the original IP problem, pygobnilp uses the Gurobi Optimizer which is a commercial optimization solver for various problems [16]. This is known as one of the fastest and most powerful

mathematical programming solvers for a linear programming problem. This algorithm will eventually return an optimal solution to the original IP problem.

1. Let x^* be the linear relaxation solution.
2. If there are cutting planes not satisfied by x^*
3. Add them and go to 1.
4. Else if x^* is integer-valued
5. The current problem is solved.
6. Else
7. Branch on a variable with non-integer value in x^* to create two new IP subproblems.

Fig. 3. Branch-and-cut approach to solving an IP problem.

In this way, pygobnilp seeks the most probable BN for the given data. It also allows users to add user constraints, such as conditional independence relations, (non-)existence of particular arrows, (non-)existence of particular undirected edges, immoralities, number of founders, number of parents, and number of arrows. The detail of the usage of pygobnilp is illustrated in its manual [17].

IV. Methods

Next, the *Conditional Gaussian (CG)* score that evaluates a BN structure is discussed in this section. This score generally calculates conditional Gaussian mixtures from the ratios of joint distributions, and it is decomposable into a sum of local scores for each parent-child relationship. Furthermore, the CG score is consistent across Gaussian mixture models because of the assumption that the data is generated from a Gaussian mixture. Additionally, it is proved to be score equivalent, which means that, if two DAGs can encode the same joint probability distribution, the scores of them are equal.

A. The method implemented in bnlearn

As a comparison, bnlearn's method to calculate the CG score is illustrated in the beginning. Bnlearn was first established by Scutari in 2007 [3], and it is an R package for learning the structure of BNs from a dataset. Although its interface is built with R, the main calculation steps, including scoring functions, are defined with C language.

Examining the C codes reveals that, when computing the CG score, bnlearn uses the conditional *maximum likelihood estimation (MLE)* in a Gaussian regression model [18]. When calculating the local CG score of a continuous variable Y with its continuous parents and discrete parents, bnlearn firstly divides the supplied data into several sub-datasets according to the settings of the discrete parents. Let the size of a sub-dataset $Data_p$ be n_p. Next, for each sub-dataset $Data_p$, bnlearn estimates a linear function of the continuous variables with a regression model. If the number of continuous parents is m, the linear function would be expressed as:

$$Y = f_p(X_1, X_2, \ldots, X_m) \tag{7}$$
$$= \beta_0 + \beta_1 X_1 + \beta_2 + X_2 + \cdots + \beta_m X_m, \tag{8}$$

where $X_i (1 \leq i \leq m)$ are continuous parents and β_i are coefficients of X_i (especially, β_0 is a parameter for a constant term). After the estimation of the linear functions for all sub-datasets, bnlearn obtains fitted values $y'_j = f_p(x_{1,j}, x_{2,j}, \ldots, x_{m,j})$ with the corresponding function f_p, where $x_{i,j}$ is the j-th observed value of the variable X_i. An observed j-th value y_j of the variable Y is assumed to be independent and normally distributed with the mean y'_j and variance σ_j^2. This variance σ_j^2 is common among the observed values that exists in the same sub-dataset, and it is estimated to be the square of the standard error s_p of the corresponding regression, which is calculated as:

$$\hat{\sigma}_j = s_p = \sqrt{\frac{\sum_{y_j \in \boldsymbol{y}_p} (y_j - y'_j)^2}{n_p - (m+1)}}, \tag{9}$$

where $\boldsymbol{y}_p$ is the vector of observed values of the variable Y that exists in the sub-dataset $Data_p$. Thus, the variance of the Gaussian distribution that the observation y_j follows is s_p^2. The probability to see the vector $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ ($n = \sum_p n_p$) in whole data $Data$ is expressed as the cumulative product of all probabilities to obtain the value y_j. Since an observation follows the Gaussian distribution $\mathcal{N}(y'_j, \hat{\sigma}_j^2)$, this can be written as the cumulative product of probability density functions of the Gaussian distributions:

$$p(\boldsymbol{y}) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\hat{\sigma}_j} e^{-(y_j - y'_j)^2 / 2\hat{\sigma}_j^2}. \tag{10}$$

Given a dataset $Data$ and the relevant vector of observations $\boldsymbol{y}$, the log-likelihood of the child variable Y is therefore computed as:

$$l(\boldsymbol{\theta}|Data) = \log p(\boldsymbol{y}) \tag{11}$$
$$= \sum_{j=1}^{n} \left\{ -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\hat{\sigma}_j^2) - \frac{1}{2\hat{\sigma}_p^2}(y_j - y'_j)^2 \right\}. \tag{12}$$

If the i-th BN variable Y_i is continuous and has both continuous and discrete parents, the local CG score $l_i(\boldsymbol{\theta}|Data)$ is calculated in this way. If all the family members are continuous variables, the log-likelihood is calculated in almost the same way, but dividing the given data is not required.

Note that a discrete variable must not have any continuous parents in bnlearn. The local score of the i-th discrete variable with its discrete parents is calculated as:

$$l_i(\boldsymbol{\theta}|Data) = \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log\left(\frac{N_{ijk}}{N_{ij}}\right) \tag{13}$$

where q_i is the number of possible configurations of the variable's parents, r_i is the number of possible values that the child variable can take, and N_{ijk} is the number of instances in which the i-th variable takes k-th value under the j-th configuration. N_{ij} represents the number of instances where the variable's parents take the j-th configuration, and thus, it is equal to $\sum_k N_{ijk}$.

After calculating the log-likelihoods of all the parent-child relationships in these ways, the CG score of a DAG is calculated by summing them up: $\sum_i l_i(\boldsymbol{\theta}|Data)$. By comparing the CG scores of possible DAGs, bnlearn heuristically search for a plausible BN structure.

However, this approach seems to have at least 2 drawbacks. First and foremost, bnlearn does not allow discrete variables to have continuous parents. Thus, this software is generally incapable of learning BNs where continuous variables influence a discrete variable. Second, due to the approach to calculate the local scores, bnlearn has to look into the given data in two phases at every calculation of a local score: when estimating the linear function; and when calculating the log-likelihood. Therefore, bnlearn appears to have flaws in adaptability and efficiency.

B. Conditional distribution and assumptions

Instead of employing the approach of bnlearn (discussed in Section IV-A), the method Andrews et al. proposed is implemented in pygobnilp. Unlike the method implemented in bnlearn, their method allows discrete children to have continuous parents, and every conditional distribution in a BN with mixed variables can be computed [11]. Suppose two continuous variables C_1, C_2 and two discrete variables D_1, D_2. If C_1 is a child with parents C_2, D_1, and D_2, the conditional distribution for C_1 is expressed as:

$$p(C_1|C_2, D_1, D_2) = \frac{p(C_1, C_2, D_1, D_2)}{p(C_2, D_1, D_2)} \tag{14}$$

$$= \frac{p(C_1, C_2|D_1, D_2)}{p(C_2|D_1, D_2)} \tag{15}$$

If D_1 is a child with parents C_1, C_2, and D_2, on the other hand, the distribution for D_1 is computed in the following way:

$$p(D_1|C_1, C_2, D_2) = \frac{p(C_1, C_2, D_1, D_2)}{p(C_1, C_2, D_2)} \tag{16}$$

$$= \frac{p(C_1, C_2|D_1, D_2)p(D_1, D_2)}{p(C_1, C_2|D_2)p(D_2)} \tag{17}$$

Each of the conditional and non-conditional probabilities in these expressions can be calculated from data. The conditional probabilities, $p(C_1, C_2|D_1, D_2)$ and $p(C_1, C_2|D_2)$, are calculated by using Gaussian distribution partitioned on the discrete variables. The computation for non-conditional ones, $p(D_1, D_2)$ and $p(D_2)$, is achieved with multinomial distribution. In general, the principles lying behind these distributions can be applied to all other cases. However, to make these probabilities possible to be calculated from data, the following assumptions are required:

1) The supplied data is generated from a Gaussian mixture.
2) The instances in the data are independent and identically distributed.
3) All Gaussian mixtures are approximately Gaussian.

A *Gaussian mixture* is a function that consists of several single Gaussian distributions $\mathcal{N}(\mu_k, \Sigma_k)$, which is defined as:

$$p(\boldsymbol{x}) = \sum_k \pi_k \mathcal{N}(\boldsymbol{x}|\mu_k, \Sigma_k) \tag{18}$$

where π_k is a probability and must meet the condition $\sum_k \pi_k = 1$ [19]. Thus, the first assumption means that the distributions of continuous variables can be expressed as a combination of several Gaussian distributions determined by conditions of discrete variables. As Figure 4 shows, for example, when the solid line represents the probability density distribution of continuous variables and two discrete variables $D_1 \in \{true, false\}$ and $D_2 \in \{true, false\}$ are involved, it is assumed that the distribution could be broken down into four Gaussian distributions, which are:

$$\begin{cases} \mathcal{N}_1 \ (\text{if } D_1 = true \wedge D_2 = true) \\ \mathcal{N}_2 \ (\text{if } D_1 = true \wedge D_2 = false) \\ \mathcal{N}_3 \ (\text{if } D_1 = false \wedge D_2 = true) \\ \mathcal{N}_4 \ (\text{if } D_1 = false \wedge D_2 = false) \end{cases} \tag{19}$$

where $\mathcal{N}_k$ denotes $\mathcal{N}(\mu_k, \Sigma_k)$. Therefore, the evaluation of a discrete variable as a child of continuous variables is achieved by the reverse relationship. This assumption promotes the efficiency of calculations.

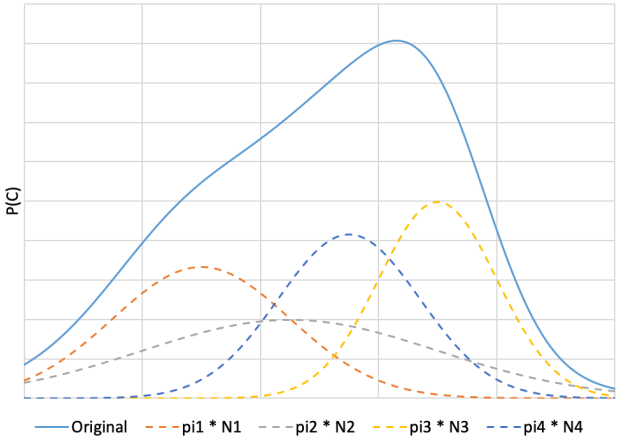

Fig. 4. Gaussian mixture of four Gaussian components. The distribution of continuous variables is assumed to follow the mixture of Gaussian distributions $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$, and $\mathcal{N}_4$.

Because of the assumption 2, the evaluation of a parent-child relationship is achieved by summing up scores calculated from data that is partitioned according to each instance of discrete variables. The detail about the calculation of scores is given in Section IV-D.

The third assumption approximates all Gaussian mixtures caused by excluding discrete variables as a single Gaussian distribution. The original distribution discussed above, for instance, is also assumed to follow a Gaussian mixture even if the discrete variable D_2 is marginalized. As is shown in Figure 5, the original distribution could be approximated as the mixture distribution that consists of two Gaussian distributions. Therefore, it could also be treated as a Gaussian distribution

$\mathcal{N}_1'$ if $D_1 = true$ and $\mathcal{N}_2'$ if $D_1 = false$. This also promotes more efficient calculation of every joint distribution in a BN.

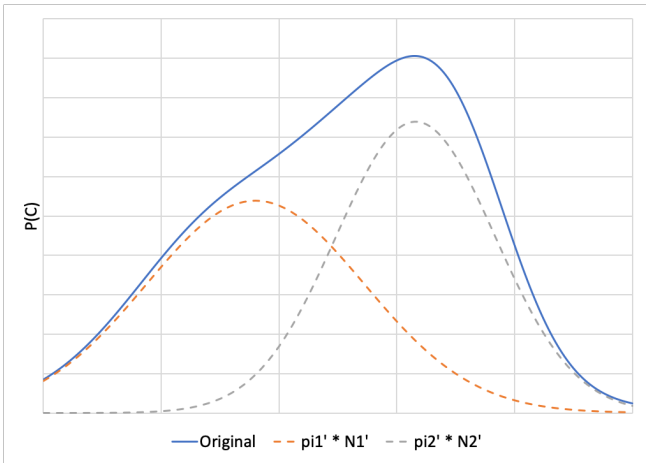

Fig. 5. The approximate Gaussian mixture components after marginalizing a discrete variable. The mixture distribution is assumed to consist of two Gaussian distributions: $\mathcal{N}_1'$ and $\mathcal{N}_2'$.

These three assumptions are significant to evaluate every kind of BN structures and to infer the most probable value of a specific variable of the BN. Due to these, the computation becomes less complicated, and as a result, evaluation can be achieved within a reasonable time.

C. Overview of CG score calculation

Before looking into the detail of the CG score calculation, its overview is provided. Let Y_i be the i-th variable in a DAG G, and suppose its parent set Pa_i. Since data is assumed to have continuous and discrete values, Pa_i can be divided into two exclusive sets: namely Pc_i that contains only the continuous parents of Y_i, and Pd_i that contains only the discrete ones, and thus, these parent subsets satisfy $Pa_i = Pc_i \cup Pd_i$ and $Pc_i \cap Pd_i = \emptyset$. To calculate the CG score of Y_i with its parents Pa_i, the method, firstly, computes the log-likelihoods and degrees of freedom for the joint distributions of $Y_i \cup Pa_i$ and Pa_i. Then, the CG score of Y_i is computed by subtracting the log-likelihood of Pa_i from that of $Y_i \cup Pa_i$. Its degrees of freedom is similarly obtained as the difference in the degrees of freedom of $Y_i \cup Pa_i$ and Pa_i.

When calculating the log-likelihood and degrees of freedom of a set of variables ($Y_i \cup Pa_i$ or Pa_i), the supplied data $Data$ is divided in the same way as bnlearn; i.e. partition $Data$ according to the settings of discrete variables involving in a variable set. Here, a partitioning set Π_i is defined according to combinations of values of the discrete variables in order to divide $Data$. Let the design matrix that is extracted by a partition $p \in \Pi_i$ be $Data_p$. $Data_p$ holds the data of continuous variables corresponding to the instances in partition p. Suppose a dataset enumerated in Table I where C_i and D_i represent continuous and discrete variables, respectively. When focusing on continuous variables C_1 and C_2 and discrete variables D_1 and D_2, the data was divided into sub-datasets according to the settings of D_1 and D_2, and an extracted sub-dataset consists of the values of C_1 and C_2. If each of D_1 and D_2 takes either of two discrete values, the partitioning set is defined as $\Pi = \{(0,0),(0,1),(1,0),(1,1)\}$. Thus, the design matrix for the partition $(0,0)$ is expressed as Table II.

TABLE I
AN EXAMPLE OF DATA. THE FIRST ROW REPRESENTS THE VARIABLE NAMES, AND OTHER ROWS ARE INSTANCES IN THE DATA. THE FIRST COLUMN REPRESENTS THE INSTANCE NUMBER.

	C_1	C_2	$\cdots$	D_1	D_2	$\cdots$
1	2.24	4.94	$\cdots$	0	0	$\cdots$
2	4.36	5.48	$\cdots$	0	0	$\cdots$
3	9.34	2.31	$\cdots$	1	1	$\cdots$
4	5.23	4.23	$\cdots$	0	1	$\cdots$
5	2.99	6.58	$\cdots$	0	1	$\cdots$
6	-1.76	1.76	$\cdots$	0	1	$\cdots$
7	5.57	5.86	$\cdots$	1	1	$\cdots$
8	5.43	8.06	$\cdots$	1	0	$\cdots$
9	-0.78	5.39	$\cdots$	0	1	$\cdots$
10	1.02	3.22	$\cdots$	0	1	$\cdots$
11	-6.00	6.76	$\cdots$	0	0	$\cdots$
12	-2.18	6.91	$\cdots$	1	1	$\cdots$
13	6.91	9.01	$\cdots$	1	0	$\cdots$
14	5.63	6.64	$\cdots$	0	1	$\cdots$
15	10.42	5.56	$\cdots$	0	0	$\cdots$
16	6.87	2.43	$\cdots$	1	0	$\cdots$
17	4.33	1.87	$\cdots$	1	1	$\cdots$
18	4.94	9.34	$\cdots$	0	1	$\cdots$
19	7.85	8.74	$\cdots$	0	0	$\cdots$
20	-3.92	7.12	$\cdots$	0	1	$\cdots$

TABLE II
THE PARTITIONED DATASET (OR DESIGN MATRIX). THIS IS GAINED BY DIVIDING THE TABLE I DATA ACCORDING TO $D_1 = 0$ AND $D_2 = 0$, AND THEN, EXTRACTING THE DATA ON C_1 AND C_2.

	C_1	C_2
1	2.24	4.94
2	4.36	5.48
11	-6.00	6.76
15	10.42	5.56
19	7.85	8.74

D. Calculating the log-likelihood and degrees of freedom

The local CG score for each variable in a DAG is calculated as follows. Suppose evaluating a specific BN variable Y_i as a child of parent set Pa_i. First of all, the log-likelihood and the degrees of freedom for $Y_i \cup Pa_i$ are calculated by summing up l_p and df_p, respectively, which are calculated separately for each partitioned dataset:

$$l_{Y_i \cup Pa_i}(\boldsymbol{\theta}|Data) = \sum_{p \in \Pi_i} l_p(\boldsymbol{\theta_p}|Data_p) \tag{20}$$

$$df_{Y_i \cup Pa_i}(\boldsymbol{\theta}) = \sum_{p \in \Pi_i} df_p(\boldsymbol{\theta_p}) - 1 \tag{21}$$

where the term minus 1 account for the redundant mixing component.

For each partition $p \in \Pi_i$, l_p and df_p are approximated in the following way. Suppose the following design matrix:

$$Data_p = \begin{pmatrix} \boldsymbol{x}_{p,1}^{\mathrm{T}} \\ \boldsymbol{x}_{p,2}^{\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_{p,j}^{\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_{p,n_p}^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & & \vdots \\ x_{j,1} & x_{j,2} & \cdots & x_{j,d} \\ \vdots & \vdots & & \vdots \\ x_{n_p,1} & x_{n_p,2} & \cdots & x_{n_p,d} \end{pmatrix} \tag{22}$$

where d is the number of variables in the design matrix $Data_p$ (or number of columns of the matrix), n_p is the number of records in $Data_p$ (or the number of rows of the matrix), and the column vector $\boldsymbol{x}_{p,j}$ denotes the j-th observation (or the transpose of the j-th row vector of $Data_p$). From this matrix, the Gaussian log-likelihood for partition p is calculated as:

$$\begin{aligned} l(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p | Data_p) = &- \frac{n_p d}{2} \log 2\pi - \frac{n_p}{2} \log |\boldsymbol{\Sigma}_p| \\ &- \frac{1}{2} \sum_{j=1}^{n_p} (\boldsymbol{x}_{p,j} - \boldsymbol{\mu}_p)^T \boldsymbol{\Sigma}_p^{-1} (\boldsymbol{x}_{p,j} - \boldsymbol{\mu}_p) \end{aligned} \tag{23}$$

where $\boldsymbol{\mu}_p$ is a column vector of means, and $\boldsymbol{\Sigma}_p$ is the covariance matrix of the Gaussian distribution. Note that $|\cdot|$ calculates the determinant of the given matrix. Here, means and covariances are assumed to be equal to the sample means and sample covariances of the partitioned data $Data_p$. Thus, $\boldsymbol{\mu}_p$ is calculated as:

$$\boldsymbol{\mu}_p = \bar{\boldsymbol{x}_p} = \frac{1}{n_p} \sum_{j=1}^{n_p} \boldsymbol{x}_{p,j}, \tag{24}$$

and $\boldsymbol{\Sigma}_p$ can be approximated as the maximum likelihood estimate $\hat{\boldsymbol{\Sigma}_p}$, which is computed as:

$$\boldsymbol{\Sigma}_p = \hat{\boldsymbol{\Sigma}_p} = \frac{1}{n_p} \sum_{j=1}^{n_p} (\boldsymbol{x}_{p,j} - \bar{\boldsymbol{x}_p})(\boldsymbol{x}_{p,j} - \bar{\boldsymbol{x}_p})^T. \tag{25}$$

Note that $\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}$ represents the outer product of vector $\boldsymbol{a}$ and $\boldsymbol{b}$. In this case, $\hat{\boldsymbol{\Sigma}_p}$ is a $d \times d$ matrix. Therefore, by replacing $\boldsymbol{\mu}_p$ and $\boldsymbol{\Sigma}_p$ with $\bar{\boldsymbol{x}_p}$ and $\hat{\boldsymbol{\Sigma}_p}$, respectively, $l(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p | Data_p)$ can be rewritten as:

$$l(\hat{\boldsymbol{\Sigma}_p} | Data_p) = -\frac{n_p}{2} (\log |\hat{\boldsymbol{\Sigma}_p}| + d \log 2\pi + d) \tag{26}$$

The log-likelihood of a Gaussian conditioned on discrete variables is calculated by the expression shown above. To calculate the desired joint log-likelihood, the log probability of an instance being from partition $p \in \Pi_i$ needs to be added to this. These probabilities are computed by the maximum likelihood estimate of variables which follow the multinomial distribution, and this is calculated as n_p/n, where n_p is the number of records in the partitioned dataset and n is the number of records of whole data. Therefore, the log-likelihood for partition p is calculated as:

$$\begin{aligned} l_p(\hat{\boldsymbol{\theta}_p} | Data_p) = &- \frac{n_p}{2} (\log |\hat{\boldsymbol{\Sigma}_p}| + d \log 2\pi + d) \\ &+ n_p \log \frac{n_p}{n} \end{aligned} \tag{27}$$

To find the number of parameters in partition p, the number of unique terms in $\hat{\boldsymbol{\Sigma}_p}$ is counted, and 1 is added to the number for the mixing component. Therefore, given a partition p, the degrees of freedom are calculated as:

$$df_p(\hat{\boldsymbol{\theta}_p}) = \frac{d(d+1)}{2} + 1. \tag{28}$$

By summing these up for all partitioned datasets, the log-likelihood and the degrees of freedom of $Y_i \cup Pa_i$ are calculated. Those of Pa_i are also computed in the same way. Finally, the local CG score and the degrees of freedom of the i-th variable Y_i with a parent set Pa_i is computed as:

$$l_i(\hat{\boldsymbol{\theta}} | Data) = l_{Y_i \cup Pa_i}(\hat{\boldsymbol{\theta}} | Data) - l_{Pa_i}(\hat{\boldsymbol{\theta}} | Data), \tag{29}$$

$$df_i(\hat{\boldsymbol{\theta}}) = df_{Y_i \cup Pa_i}(\hat{\boldsymbol{\theta}}) - df_{Pa_i}(\hat{\boldsymbol{\theta}}). \tag{30}$$

In this way, each variable's parent-child relationship is evaluated. After computing these local CG scores for each variable, this method calculates the overall evaluation of a DAG given a dataset by summing up all the local scores.

The score is often penalized according to the complexity of the BN because, without any penalties, the high-scored BN tends to have as many arrows as possible. A popular approach to add a penalty to the score takes the form of $\phi(G|Data) = LL(G|Data) - f(N)|G|$ where $f(N)$ is a non-negative penalization function of the number of instances and $|G|$ represents the complexity of the BN [7]. Considering the complexity as the degrees of freedom, the penalized scoring function is rewritten as:

$$\phi(G|Data) = \sum_i l_i(\hat{\boldsymbol{\theta}} | Data) - f(N) \sum_i df_i(\hat{\boldsymbol{\theta}}). \tag{31}$$

An example of such a function is called *Akaike Information Criterion (AIC)* [20] where $f(N) = 1$. If $f(N) = \frac{1}{2} \log N$, we have the *Bayesian Information Criterion (BIC)* score proposed by Schwarz [21]. $f(N) = 0$ means no penalties, and thus, the score is just the normal CG score.

By calculating the CG score in this way and comparing them, pygobnilp exactly learns the most probable BN that gained the highest score.

This approach calculates essentially the same scores as bnlearn. However, this may be better than bnlearn because of its broader utility and greater potential to optimize the program. As is illustrated above, the approach can evaluate a discrete variable with continuous parents. Thus, implementing this method, a search algorithm can examine all possible BN structures.

Furthermore, since this method calculates the CG score based on covariance matrices, the calculation may become more efficient by storing covariance matrices, because it remains consistent given a dataset. Particularly, if all the BN

variables are continuous, the covariance matrix of all variables calculated from whole data can be reused whenever calculating a local non-conditional Gaussian score. For any subsets of variables, it is enough to compute the sample covariance matrix for all continuous variables just once and then extract sub-matrices to compute the log-likelihood.

Each computed log-likelihood is additionally worth storing because a local CG score is calculated from log-likelihoods of two variable sets (namely set of parents and set of child + parents). For instance, when naively calculating the local CG scores of two parent-child relationships, $\{A, B\} \to C$ and $\{A, B\} \to D$, it is necessary to calculate four log-likelihoods: $l_{\{A,B\}}$ and $l_{\{A,B,C\}}$ for evaluating $\{A, B\} \to C$; and $l_{\{A,B\}}$ and $l_{\{A,B,D\}}$ for evaluating $\{A, B\} \to C$. However, by remembering log-likelihoods, program is required to calculate $l_{\{A,B\}}$ only once. Therefore, the CG scoring method proposed by Andrews et al. may still have a room to be optimized, and it may become more useful and efficient than bnlearn.

V. Experiments

This section describes a series of experiments that were conducted to validate the implemented method. The new methods which calculate the CG score were coded with Python and embedded in pygobnilp. Through the experiments, the functionality of pygobnilp with this method was assessed.

First, the local CG scores computed by pygobnilp were compared with those of bnlearn. As both methods calculate the identical values, the results when using pygobnilp were expected to be nearly equal to the score calculated by bnlearn. This experiment would corroborate that the implemented method performs the correct calculation.

Second, the time that pygobnilp and bnlearn took to calculate the Gaussian scores or CG scores of a specific number of BNs from data was measured for different data sizes. By storing log-likelihoods for each variable set and whole covariance matrices, pygobnilp was predicted to be more efficient than bnlearn. Through this experiment, the implemented approach is be evaluated in terms of efficiency.

Third, the calculation time to obtain the local CG scores of all possible parent-child relationships was measured. In this experiment, the number of BN variables was varied, whereas the data size was fixed. The calculation time may be slowed down exponentially because the number of possible relationships to be scored will rise exponentially as the number of variables increases. This experiment would reveal how many variables pygobnilp can deal with within a reasonable time.

Finally, the precision of BNs that were learned by pygobnilp or by bnlearn was measured. The performance of pygobnilp is expected to be better than bnlearn because pygobnilp can calculate the CG score of the relationship where a discrete variable has continuous variables as its parents while bnlearn cannot. The question whether the implemented method enables pygobnilp to learn more generic BN from data would be addressed by this experiment.

In these experiments, a laptop whose operating system was macOS Catalina was used. The processors consist of 1.2GHz quad-core Intel Core i7, and the memory size was 16GB. The experiments were conducted on the anaconda platform whose version was 1.7.2. The programs of pygobnilp were run by Python 3.7.7, and bnlearn was run by R 3.6.1.

A. Result of calculating conditional Gaussian scores

In the first experiment, local CG scores for 95 families were calculated with pygobnilp and bnlearn from a dataset called "clgaussian.test", which is provided by bnlearn packages. This dataset consists of 4 discrete variables (A, B, C, F) and continuous variables (D, E, G, H), and it has 5,000 instances.

Table III enumerates the 12 out of 95 randomly generated families and their local CG scores. Among all the calculated local scores, the greatest difference between pygobnilp and bnlearn was 0.127, and the highest relative difference was 0.007%, both of which were observed when evaluating the parent-child relationship $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\} \to \mathrm{D}$.

TABLE III
THE LOCAL CG SCORES FOR 12 FAMILIES CALCULATED FROM CLGAUSSIAN.TEST. A FAMILY $\{\mathrm{X}, \mathrm{Y}\} \to \mathrm{Z}$ MEANS THAT CHILD Z HAS $\{\mathrm{X}, \mathrm{Y}\}$ AS ITS PARENT SET.

Family	pygobnilp	bnlearn	difference
$\emptyset \to$ A	-1567.524	-1567.524	0.000
$\{$A, B$\} \to$ D	-1757.816	-1757.822	0.006
$\{$A, F$\} \to$ D	-1760.120	-1760.122	0.002
$\{$A, B, C$\} \to$ D	-1738.733	-1738.860	0.127
$\{$B, F, G$\} \to$ D	1510.355	1510.346	0.009
$\{$A, B, F$\} \to$ E	-7933.648	-7933.676	0.028
$\{$B, D, F$\} \to$ E	-6531.378	-6531.387	0.009
$\{$A, B, F$\} \to$ G	-11444.497	-11444.525	0.028
$\{$A$\} \to$ H	3447.735	3447.735	0.001
$\{$A, C, D$\} \to$ H	3780.858	3780.809	0.049
$\{$A, B, D$\} \to$ H	3784.368	3784.343	0.025
$\{$A, E, G$\} \to$ H	3463.099	3463.094	0.005

Since the differences between pygobnilp and bnlearn were relatively small, they may be caused by rounding errors. This small difference may also be because of the difference in calculating a variance; the implemented method calculated a sample variance, whereas bnlearn adopted an unbiased variance (see Section IV-A and IV-D). Therefore, the results of this experiment could draw the conclusion that the implemented method can correctly calculate the CG scores.

B. Result of measuring calculation time for different sample size

Next, the calculation time of the CG scoring function was measured. In this experiment, 1,000 BNs, which were learned with pygobnilp from a mixed dataset, "clgaussian.test", were evaluated with bnlearn and two versions of the implemented scoring method; one calculated the log-likelihoods of variable sets every time, and the other stored and reused them. These methods calculated the CG scores of the BNs from datasets, which had 5,000 to 2,560,000 instances and were derived from a mixed dataset "clgaussian.test". Each dataset was generated by repeating "clgaussian.test" either once, twice, four times, ..., or 512 times.

Figure 6 represents the calculation time of the three different CG scoring functions. As a general trend, the time was proportional to the data size for each method. Categorically speaking, the pygobnilp method that stored the log-likelihoods was the fastest. Therefore, storing the log-likelihoods for each variable set was revealed to be effective. However, the method that calculates log-likelihoods every time was slower than bnlearn. The reasons for this might be that bnlearn used the compiled C programs while pygobnilp was not compiled, and that both looked into the given data twice when calculating a local CG score of one parent-child relationship.

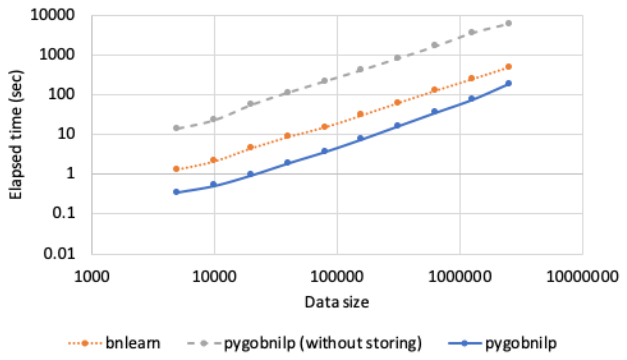

Fig. 6. The comparison of calculation time of the CG scoring functions of bnlearn, pygobnilp that does not store the log-likelihoods, and pygobnilp that remembers the log-likelihoods. Note that this is a double logarithmic plot.

The time of calculating the Gaussian scores was measured identically. The datasets were generated from "gaussian.test", which is a continuous dataset provided by bnlearn, and the data size was varied from 5,000 to 2,560,000. Each dataset was similarly generated by concatenating 1, 2, 4, ..., or 512 identical datasets, "gaussian.test". The Gaussian scoring methods in this experiment were 1) the bnlearn method, 2) the pygobnilp method that stores whole covariance matrix, and 3) the pygobnilp method that stores both whole covariance matrix and log-likelihoods.

Figure 7 illustrates the results of measuring the calculation time of the Gaussian scores. Again, this figure suggested the positive effects of storing log-likelihoods. Notably, even the pygobnilp method that stored only the whole covariance matrix was subsequently faster than the bnlearn method. This would primarily because the program was required to look into the given dataset only once when the whole covariance matrix was stored and reviewed to calculate log-likelihoods.

Examining Figure 6 and 7, calculating the CG scores took substantially longer than the Gaussian score. The program has to divide the given data when calculating the CG score, whereas the calculation of the Gaussian score does not require this partitioning. Because of this difference, the calculation of scores would be slowed down when mixed data was supplied. Furthermore, pygobnilp has to calculate the covariance matrix for each partitioned dataset. Thus it can be considered as another factor of the increase in the calculation time of the CG score compared to that of the Gaussian score.

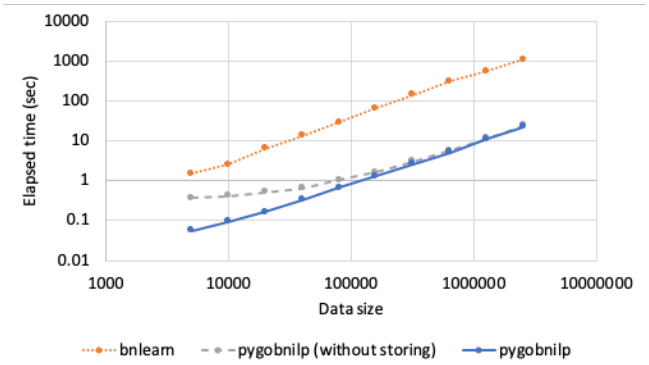

Fig. 7. The comparison of calculation time of the Gaussian scoring functions of bnlearn, pygobnilp that does stores the whole covariance matrix, and pygobnilp that stores the whole covariance matrix and log-likelihoods. Note that this is a double logarithmic plot.

C. Result of measuring calculation time for different number of variables

Additionally, the time to calculate the CG scores of all the possible parent-child relationships was measured for the different numbers of variables. The test data was randomly generated with a desktop Java application, tetrad [4], which offers many functions for causal and statistical models, including generating random BNs and simulating a BN to generate various data. First, random BNs with 8, 10, 12, 14, 16, 18, and 20 variables were built, and then, 7 test datasets with 5,000 records were generated based on these BNs. For each dataset, the number of discrete variables were exactly equal to that of continuous variables, and the maximum number of possible values that discrete variables can take was set to three. In this experiment, the method for calculating the CG scores stored and reused the previously calculated log-likelihoods of variable sets.

Figure 8 represents the result of measurement of calculation time for each dataset. The line graph shows the exponential increase in the calculation time, and with 20 variables, the calculation costed over 5 hours.

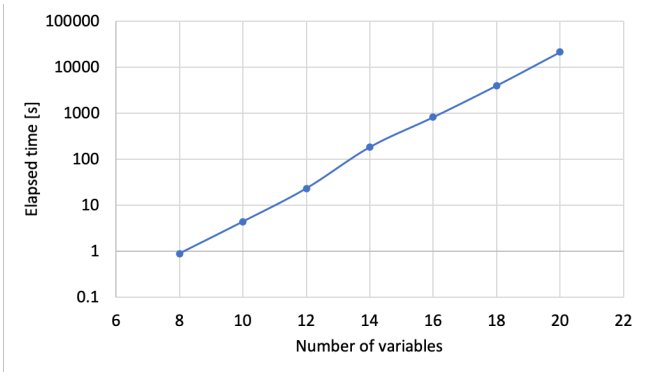

Fig. 8. The change in the calculation time of the CG scoring functions with respect to the number of variables. Note that the elapsed time was displayed on a logarithmic scale.

This tendency would be natural because the numbers of

variable sets and possible families are exponentially increased by the number of variables. If the number of BN variables is $|V|$, pygobnilp is required to calculate log-likelihoods for $2^{|V|} - 1$ variable sets in order to obtain the CG scores of possible $|V| \times 2^{|V|-1}$ families. Therefore, pygobnilp is estimated to cost an exponentially longer time to learn a BN from the data with more variables.

D. Result of learning a Bayesian network from data

Finally, in order to evaluate the implemented method in terms of precision, a BN was learned from a dataset, and it was compared with the true BN that generated the dataset. In this experiment, a dataset with 5,000 observations was randomly generated by simulating a 20-variable BN with tetrad. This BN was randomly generated by tetrad, and then each variable was designated as either continuous or discrete. In order to simplify the experiment, the maximum number of parents of a BN variable was set to three, and the maximum number of categories that a discrete BN variable can take was set to three. Exactly half of the BN variables were designated as discrete and others were continuous. The randomly generated BN consequently had 37 directed edges, and variable names and their parents are illustrated in Table IV.

TABLE IV
LIST OF VARIABLES OF THE RANDOMLY GENERATED BN AND THEIR PARENTS. IN "TYPE" COLUMN, "C" IS DENOTED AS "CONTINUOUS" AND "D" MEANS "DISCRETE".

Variable	Type	Parents	Variable	Type	Parents
X1	C	$\emptyset$	X11	D	X2, X3
X2	D	$\emptyset$	X12	C	X7, X8
X3	C	$\emptyset$	X13	C	X6, X11, X12
X4	D	X3	X14	C	X7, X11, X13
X5	C	X1, X2	X15	C	X1, X12, X14
X6	D	X1, X4	X16	D	X5, X6, X10
X7	D	$\emptyset$	X17	D	X2, X9, X14
X8	C	X5	X18	D	X3, X7, X16
X9	D	X5, X6	X19	C	X18
X10	D	X4, X8, X9	X20	C	X4, X8, X12

Given the dataset extracted from this BN, pygobnilp and bnlearn learned a BN, and afterward, the *Structural Hamming Distance (SHD)* was calculated between that and the true BN. Briefly speaking, SHD between a learned BN and a true BN is the number of edges that need to be added, deleted, or flipped to make the two BNs identical (the detailed description was provided by Tsamardinos et al. [22]). Thus, if SHD is 0, the learned BN is the same as the true BN, and the greater SHD becomes, the more different the learned BN is concluded to be from the true BN.

Table V enumerates the SHDs of the BNs learned by either pygobnilp or bnlearn. During the learning process, pygobnilp and bnlearn used either the CG, BIC, or AIC scoring function. From this table, the BN that was learned by pygobnilp with the BIC score gained the smallest SHD. The SHD was smaller when the BIC or AIC score was used to evaluate a BN compared to when the program used the CG score. Further, the SHDs of the BNs learned by pygobnilp was smaller than those learned by bnlearn for each scoring method.

TABLE V
THE SHDS BETWEEN A LEARNED BN AND THE TRUE BN. BNS WERE LEARNED WITH PYGOBNILP OR BNLEARN, AND THE CG, BIC, AIC SCORES WERE USED TO SCORE A NETWORK.

	pygobnilp	bnlearn
CG	23	31
BIC	6	18
AIC	16	24

Since the SHD of a BN was smaller when a penalized scoring function was used, penalizing according to the complexity of a BN would result in more precise prediction of a BN structure. More importantly, pygobnilp was able to learn a BN that was closer to the true BN than bnlearn. This may be because bnlearn cannot calculate the log-likelihood of a discrete variable with continuous parents.

As mentioned in Section IV, pygobnilp exactly search for the most probable BN, whereas bnlearn employs a heuristic learning approach. Thus, to clarify the influence of a learning approach, a new dataset was used that is generated from a BN where a discrete variable must not have a continuous variable as its parent. First, a BN structure was randomly generated by tetrad, and then, each variable type was manually designated to make sure that a discrete variable does not have a continuous parent. After that, the parameter set was defined randomly, and the generated model was simulated to obtain a dataset. To make the experiment simple, the number of categories that each discrete variable can take was set to two. The randomly generated BN had 44 directed edges, and variable names and their parents are illustrated in Table VI.

TABLE VI
LIST OF VARIABLES OF THE BN AND THEIR PARENTS WHERE NO DISCRETE VARIABLES HAVE A CONTINUOUS PARENT. IN "TYPE" COLUMN, "C" IS DENOTED AS "CONTINUOUS" AND "D" MEANS "DISCRETE".

Variable	Type	Parents	Variable	Type	Parents
X1	D	$\emptyset$	X11	C	X2, X3
X2	D	X1, X5, X12	X12	D	X5, X7, X19
X3	C	X5, X9, X18	X13	D	X8, X19
X4	C	X10, X17	X14	C	X1, X4
X5	D	X1	X15	C	X7, X9
X6	C	X5, X9, X13	X16	C	X1, X4
X7	D	X5, X9, X10	X17	C	X1, X9
X8	D	X5, X7, X10	X18	C	X5, X16
X9	D	X1, X5	X19	D	X8, X10
X10	D	X1, X5, X9	X20	C	X10, X17

Table VII shows the SHDs calculated from the learned BNs. BNs were learned according to either the CG, BIC, or AIC score. When learning a BN structure with pygobnilp, the constraints that arrows from a continuous variable to a discrete variable must not exist were added. From this table, the BNs that were learned by pygobnilp with the CG and BIC score gained the smallest SHD. The overall result was similar to that of the previous experiment; the SHDs of the BNs learned by pygobnilp were smaller than those learned by bnlearn. However, the SHDs of BNs that were learned according to AIC were greater than those of other scores.

Therefore, pygobnilp were able to learn the BNs that was

TABLE VII
THE SHDS BETWEEN A LEARNED BN AND THE TRUE BN THAT DOES NOT ALLOW ANY DISCRETE VARIABLES TO HAVE A CONTINUOUS PARENT. BNS WERE LEARNED WITH PYGOBNILP OR BNLEARN, AND THE CG, BIC, AND AIC SCORING FUNCTIONS WERE USED.

	pygobnilp	bnlearn
CG	21	34
BIC	21	29
AIC	29	35

closer to the true one than bnlearn did. The reason for this similar tendency may be that bnlearn could not find the optimal BN with heuristic approach.

In Table VIII, the CG, BIC, and AIC scores of all the learned BNs were listed. Each cell in the table shows the score of a different learned BN, while every score was computed by bnlearn. Thus, the score in "BIC" row and "pygobnilp" column, for example, shows the BIC score of the BN that is learned by pygobnilp with the BIC scoring function. This table explicitly suggests that each of the scores of the BNs learned by pygobnilp was greater than those of the BNs learned by bnlearn, respectively.

TABLE VIII
THE CG, BIC, AND AIC SCORES OF ALL THE LEARNED BNS.

	pygobnilp	bnlearn
CG	-135377.9134	-136199.8567
BIC	-136013.411	-136844.2202
AIC	-135572.5849	-136389.0865

As Table VIII shows, bnlearn did not find the BN that gained the highest score. Thus, it was elucidated that the heuristic approach influences the precision of a learned BN.

In this experiment, pygobnilp showed a higher performance in learning a general BN compared to the precision of bnlearn. The difference in precision between heuristic approach and exact approach was also clarified; even if a BN does not include any arrows from a continuous variable to a discrete variable, the BNs learned by pygobnilp were more similar to the true BN than that of bnlearn. Therefore, pygobnilp was revealed to be more useful in terms of precision.

E. Limitation

Since bnlearn cannot calculate the log-likelihood of a specific parent-child relationship without a concrete BN, it was not possible to compare the calculation time of pygobnilp with that of bnlearn by calculating the score of arbitrary families. Further, this made it impossible to calculate all the possible families in a BN, and as a result, the relationship between calculation time and the number of variables was not able to be measured when bnlearn was used.

VI. CONCLUSION

In this paper, an approach to calculate the conditional Gaussian score, which was proposed by Andrews et al [11], was illustrated and practically implemented in pygobnilp. Briefly describing, the method calculated the CG score of a BN by summing up the local CG scores of the parent-child relationships. The local CG score was computed by 1) partitioning the given data, 2) calculating the covariance matrices for each sub-dataset, 3) calculating the log-likelihoods of the set of the parents and that of the child plus parents, and 4) subtracting the log-likelihood of the set of the parents from that of the set of the child plus parents.

The series of experiments suggested that the implemented method calculated the CG score correctly. Moreover, by making the method remember the calculated log-likelihoods of variable sets, it was revealed that the calculation of the CG score would be more efficient than bnlearn. When data had only continuous values, storing the whole covariance matrix also made the calculation considerably faster. Thus, optimization of the method to calculate the CG score was revealed to be highly effective compared to the method implemented in bnlearn. Since the calculation of the CG score was exponentially slowed down as the number of variables increased, learning a BN with pygobnilp from data with hundreds of variables may be significantly hard. However, pygobnilp may be more useful to various kinds of data because the BN learned by pygobnilp was closer to the true BN than that of bnlearn. This result would be because of the difference in learning approach; pygobnilp adopted an exact learning method, whereas bnlearn employed a heuristic one. Through these experiments, therefore, the implemented method was revealed to be effective.

In future work, it should be ensured that the implemented method can calculate the local CG score of a discrete child with continuous parents because this has not been confirmed in the experiments. The accuracy of learning a BN structure should be assessed by using more various datasets to ensure whether pygobnilp can learn a wider variety of BNs. These further evaluations would bolster the effectiveness of the implemented method. Additionally, since it takes too much time to calculate the CG score from the data with many variables, optimization of pygobnilp may be necessary to avoid or quicken the calculation of many local scores. It may also be interesting to invent the way to exploit the covariance matrices which are previously calculated during the computation of the local CG score. As the method currently looks into the given data twice to calculate a local CG score, this optimization would make the method more efficient.

ACKNOWLEDGMENT

I would like to thank Mr. James Cussens for guiding me to many relevant publications, for advising on the tasks to do, and for resolving many questions on Bayesian networks. The meetings and conversations were vital in clarifying unsure points and inspiring me to think of the project more deeply.

REFERENCES

[1] W. Wiegerinck, et al., "Bayesian networks for expert systems: theory and practical applications," in *Interactive collaborative information systems*, Springer, 2010, pp. 547-578.

[2] M. Barlett and J. Cussens, "Advances in Bayesian Network Learning Using Integer Programming," *ArXiv.org*, March 23, 2015.

[3] M. Scutari. *bnlearn - Bayesian network structure learning*, bnlearn.com. [Online]. Available: https://www.bnlearn.com/. [Accessed: 21 June 2020].

[4] C. Glymour, et al. (2015). *Tetrad*, Carnegie Mellon University. [Online]. Available: http://www.phil.cmu.edu/tetrad/. [Accessed: 21 June 2020].

[5] University of York (2017, Jan.). *GOBNILP*, cs.york.ac.uk. [Online]. Available: https://www.cs.york.ac.uk/aig/sw/gobnilp/. [Accessed: 21 June 2020].

[6] J. Cussens, "Integer Programming for Bayesian Network Structure Learning," *Quality Technology & Quantitative Management*, vol. 11, no. 1, pp. 99–110, 2014.

[7] A. M. Carvalho, "Scoring functions for learning Bayesian networks," INESC-ID Tec, Rep. 54/2009, April 2009.

[8] S. S. Qian and R. J. Miltner, "A continuous variable Bayesian networks model for water quality modeling: A case study of setting nitrogen criterion for small rivers and streams in Ohio, USA," *Environmental Modelling & Software*, vol. 69, pp. 14-22, 2015.

[9] Z. Liu, et al., "Empirical Evaluation of Scoring Functions for Bayesian Network Model Selection," *BMC Bioinformatics*, vol. 13, no. Suppl 15, 2012.

[10] Y. C. Chen, et al., "Learning discrete Bayesian networks from continuous data," *Artificial Intelligence Research*, vol. 59, pp. 103-132, 2017.

[11] B. Andrews, et al., "Scoring Bayesian networks of mixed variables," *International Journal of Data Science and Analytics*, vol. 6, no. 1, pp. 3–18, 2018.

[12] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Royal Statistical Society: Series B (Methodological)*, vol. 50, no. 2, pp. 157-294, January 1988.

[13] H. Pishro-Nik (2014), *Conditional independence.* Kappa Research LLC [Online]. Available: https://www.probabilitycourse.com/chapter1/1_4_4_conditional_independence.php. [Accessed: Aug. 17, 2020].

[14] R. Trotta, "Bayes in the sky: Bayesian inference and model selection in cosmology," *Contemporary Physics*, vol. 49, no. 2, pp. 71-104, 2008.

[15] T. Jaakkola, et al., "Learning Bayesian network structure using LP relaxations." in *Proceedings of the Thirteenth Informational Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, Italy, 2010. pp. 358-365.

[16] Gurobi. *Gurobi - The fastest solver*, gurobi.com. [Online]. Available: https://www.gurobi.com/products/gurobi-optimizer/ [Accessed: Aug. 7, 2020].

[17] *pygobnilp manual (version 1.0)*, J. Cussens, University of York, January 30, 2020.

[18] M. Lavielle (2016, Nov. 30). *Maximum likelihood estimation in a Gaussian regression model*. Statistics in Action with R [Online]. Available: http://sia.webpopix.org/regressionML.html. [Accessed: 29 May 2020].

[19] O. C. Carrasco (2019, Jun. 3). *Gaussian mixture models explained.* Towards Data Science [Online]. Available: https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95. [Accessed: Aug. 24, 2020].

[20] H. Akaike, "A new look at the statistical model identification," *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716-723, December 1974.

[21] G. Schwarz, "Estimating the dimension of a model," *Annals of Statistics*, vol. 6, no. 2, pp. 461–464, 1978.

[22] I. Tsamardinos, et al., "The max-min hill-climbing Bayesian network structure learning algorithm," *Machine learning,* vol. 65, no. 1, pp. 31-78, 2006.