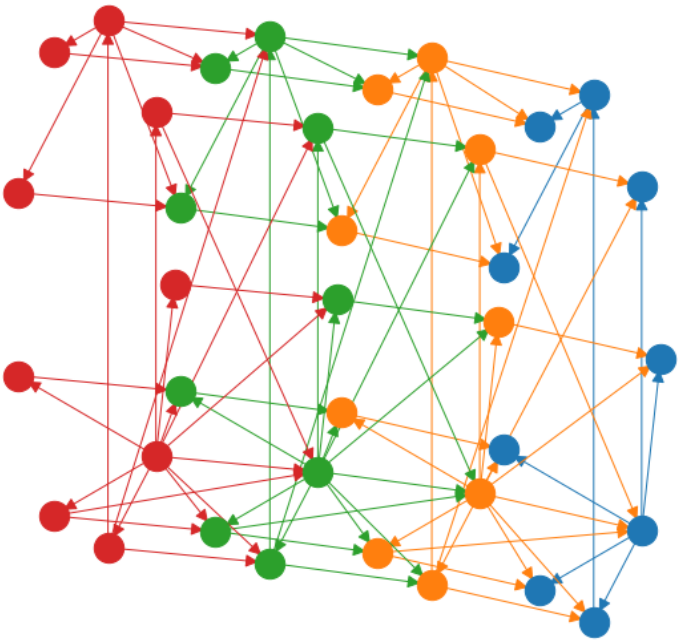
Master Thesis: Learning Classification-DBNs from Data



Author:
Daan Knoope, Universiteit Utrecht, d.a.s.knoope@uu.nl

First Supervisor:
Dr.habil. Cassio P. de Campos, Universiteit Utrecht, c.decampos@uu.nl

Second Supervisor:
Dr. S. Renooij, Universiteit Utrecht, s.renooij@uu.nl

Daily Supervisor:
Ir. Wouter Poncin, de Volksbank, wouter.poncin@devolksbank.nl

ICA-4102622

Utrecht University

August 9, 2019

Abstract

As our societies are becoming ever more digital and reliant on automated systems, it becomes increasingly important to monitor the technologies we depend on using automated systems to guard against failures and downtime. While many fault detection solutions have already been proposed, we found that methods for continuously monitoring the state of a system in an explainable way have not yet been widely researched, while this could provide helpful information to the user. Therefore, we propose C-DBNs, a special case of Dynamic Bayesian networks that have been tailored to classify dynamic processes using existing probabilistic models. We also introduce S-RAD: a novel method for automatically discretizing datasets for usage with C-DBNs to automate the process of learning explainable models even further. Our first results seem promising and provide a reliable alternative to existing methods of discretization without prior knowledge.

Contents

Chapter 1

Introduction

As our societies are becoming more digital and reliant on automated systems, it becomes increasingly important to monitor the technologies we depend on. With rising amounts of data, it has become untenable for humans alone to take on this task. Therefore, much research has been done towards finding machine learning solutions that help us analyze patterns and spot potential problems [9,16,19,31]. One interesting domain is that of monitoring systems over time and classifying their status. Examples of this could include predicting dangerous weather conditions, spotting erroneous behavior in servers, and classifying road conditions. These problems all involve a temporal component in which variables gradually change, and an entire situation needs to be classified. In the context of monitoring systems, some human intervention or change of behavior is often required upon reaching a certain classification. It can, therefore, be helpful to have a system that can explain its predictions and provides a certain amount of reasoning for the decision that it has taken. This explanation could allow the user not only to check the reasoning process but also attain more context so they can react faster.

Before writing this thesis, we were unable to find existing research that combines the idea of continuously classifying time-series in doing this an explainable fashion. It is, therefore, the aim of this thesis to describe an approach of how such a system can be implemented. For this, we use Bayesian networks as an underlying model. BNs can reason with uncertainty using probabilities and produce a graphical model in which relationships between different variables are shown. These networks can be queried for alternative situations as well as the certainty of its decisions, which both provide a lot of opportunity for creating an explainable system.

The creation of a Bayesian network for monitoring continuous systems is not trivial, however. First, there needs to be a way for these networks to handle temporal data. While this has already been studied extensively in the form of Dynamic Bayesian networks [32], these are not explicitly targeted towards classification. Since we want to create a solution that is fully dedicated to classification, we will introduce the concept of Classification-Dynamic Bayesian Networks (C-DBNs). In addition, while there are solid toolboxes for learning regular Bayesian networks from data [4, 12], this is not the case for their dynamic variants. To solve this issue, we propose a formal and practical description of how the problem of learning a DBN can be transformed into an instance of BN structure learning and parameter estimation.

Learning a C-DBN can, however, sometimes still require manual work, even with the transformation in place. This work consists of the discretization of continuous real-world values. To make the process of learning as automated as possible, we introduce a method of pushing the task of discretization into the structure learning process. We first show how this can be done in an optimal, but infeasible way, which we call Full Automatic Discretization, or FAD. We then

reduce the complexity of this method using heuristics, in an approach we call Selection-based Reduced Automatic Discretization, or S-RAD. Our proposed method of automatic discretization is a stable and safe alternative to other methods of discretization in which no prior knowledge is assumed. We also make some suggestions about how our current approach can be expanded upon.

To cover these topics, we will, in the next chapter, provide a concise overview of the theory of Bayesian networks and discretization. Then, we introduce our proposal for C-DBNs, formalize it, and compare it against a similar and already existing method. In the following chapter, we describe how (C-)DBNs can be learned from data using existing toolkits for Bayesian networks. Having finished the first part of the thesis, we will describe the optimal but infeasible approach to automatic discretization for C-DBNs, FAD, in Chapter 5. We then introduce S-RAD, the heuristic-based variant of automatic discretization in chapter 6. Finally, we test our approach to C-DBNs and S-RAD in Chapter 7, analyze the results in Chapters 8 and 9, and conclude with suggestions for further research.

Chapter 2

Theory and Related Work

2.1 Bayesian networks

A Bayesian Network (BN) $\mathcal{G} = (V, E, \Theta)$ is a probabilistic graphical model consisting of a tuple containing a Directed Acyclic Graph (DAG) specified by a set of vertices V, a set of directed edges E, and a joint probability distribution Θ [33, p. 45]. The vertices represent variables, while the edges indicate dependencies between the variables. Variables can take on discrete values, although alternative versions which allow continuous values exist [26].

We use the graph of a Bayesian network to encode independencies between the variables. There are two types of independencies that we can show: direct independencies of the form $A \perp B$, where A is independent of B, and conditional independencies, $A \perp B|C$, where A and B are independent given C. These conditional independencies help us create relationships between the variables. A direct independency could, for example, be the performance of a network switch and the number of goals scored by a local soccer club: the two are entirely unrelated. A conditional independency is more subtle: it might provide us with information, but there is no direct relationship. The performance of a network switch could, for example, be conditionally independent of whether there is currently a thunderstorm raging. A thunderstrike might cause a power surge, causing the switch to turn itself off. However, if we also monitored the power grid and knew there was no power surge, knowing the current weather is of no importance to us. Thus, thunderstorms and switches can be conditionally independent. We can represent this information in a Bayesian network, as shown in Figure 2.1.

Using these independencies, we can create the joint probability distribution Θ in a succinct way. We can decompose the joint distribution by chaining the conditionally independent probabilities [28, p. 61]. In our example, we can decompose the distribution of our entire model into smaller

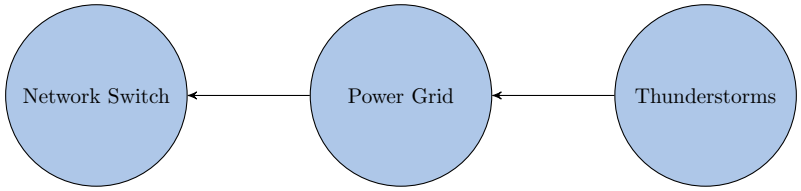


Figure 2.1: Example of a BN with $V = \{\text{Network Switch, Power Grid, Thunderstorms}\}$ and $E = \{(\text{Thunderstorms}, \text{Power Grid}), (\text{Power Grid}, \text{Network Switch})\}$

parts: $P(N, P, T) = P(N|P)P(P|T)P(T)$ (with N for network switch, P for power grid, and T for thunderstorms).

The resulting model generally has a smaller complexity than alternatives without (conditional) independencies and can be used for reasoning with probabilities. We can use a Bayesian network for various tasks, including finding the probability of events given a set of evidence. If we want to use a BN for classification, we can use a technique called Maximum A-Posteriori querying (MAP-queries), which tries to find the values for the variables that maximize the posterior probability given the available evidence. Due to its graphical nature and flow of evidence, Bayesian networks are a form of artificial intelligence that is well suited to show both *why* classifications have been made, and how *certain* it is about these predictions.

2.2 Learning from data

In the past, Bayesian networks could only be constructed by knowledge elicitation from domain experts [29, p. 297]. In these methods, there was often a trade-off between time consumption and precision [39]. With the introduction of data mining, techniques have been developed to learn Bayesian networks from data, rather than by knowledge elicitation. This allows us to learn models automatically when we have enough data at our disposal.

The task of learning BNs from data can be split into two distinct processes: parameter estimation and structure learning. In parameter estimation, we want to learn the joint probability distribution for the network. There are two ways to do so: by maximum likelihood estimation, and by using Bayesian approaches [28, p. 717]. In maximum likelihood estimation, we want to maximize the probability of seeing the dataset we have given our parameters. Since the likelihood score decomposes [28, p. 723], we can do so by choosing the probabilities for variables by counting the frequency at which they occur in the dataset. This method only relies on the exact data we have seen. Bayesian approaches add an extra component to this: a prior which tends to decline in power as we see more data points. Using a prior allows us to either add evidence or use an uninformative (i.e., uniform) prior. The uniform prior can also function as regularization against overfitting.

Besides obtaining the probability distributions from data, we can also learn the structure, i.e., the graph, of the Bayesian network. In its simplest form, we can again use maximum likelihood estimation for this: we can loop over all the possible edge combinations and find the one for which we can achieve the best fit for the parameters. As with parameter estimation, we can also use a version of structure learning that allows us to specify a prior on the range of possible graphs. For this, we often use the BDeu score, which is an uninformative and uniform prior for the structure learning process which assigns networks that represent equivalent joint probability distributions the same score [21].

2.3 Dynamic Bayesian networks

Dynamic Bayesian Networks (DBNs) are an extension of Bayesian networks in which variables can exist multiple times in the network [32]. This multiplicity of vertices is used to track the progression of variables over time [29, p. 112]. A DBN consists of multiple time-slices or time-steps, in which each time-step consists of its own full Bayesian network. The time-steps are connected by edges that symbolize the causal relationship between variables over time. We can limit the number of time-frames an edge can skip over to represent the Markov-assumption:

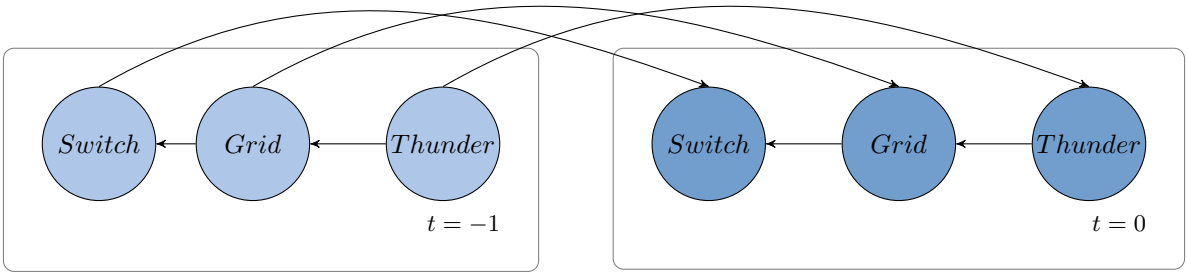


Figure 2.2: Example of Figure 2.1 represented as a DBN

the idea that the future is conditionally independent of the (distant) past, given the present. DBNs allow us to reason about processes that change over time. Going back to our example of the network switch, we could use DBNs to model a power grid. If we see the status of the grid repeatedly changing over time, we might be able to make a more accurate guess about the status of our switch than if we *only* knew the grid was operating correctly at one specific moment. This is shown in Figure 2.2.

Besides this example, Dynamic Bayesian networks have been used in many real-world applications. They have primarily been used for reasoning under uncertainty. Chen et al. [10], for example, used DBNs to support risk-informed decisions when dealing with flood-control operations. Wu et al. used DBNs for modeling risk in tunnel constructions [43], while Hofleitner et al. used DBNs to create a statistical model of traffic times based on sparse (and incomplete) probe data [23], and Khakzad recently used them to model wildfire spread [27]. Another, perhaps less conventional reason to use DBNs, is to understand or explain an underlying model, due to the same graphical nature and flow of evidence DBNs have in common with regular BNs. Tucker et al. for example, used a type of DBN to analyze differences in biological mechanisms [42].

In this thesis, we want to use the foundations of DBNs to predict classification labels in time-series because of their explainable nature. Since we do not want to construct models by knowledge elicitation, part of this project will be to describe how C-DBNs can be learned from data using existing and well-studied toolboxes for regular Bayesian networks. By following this approach, we can utilize well-researched strategies and software.

2.4 Discretization

While Bayesian networks can, as noted before, use continuous data, this is not entirely common. Especially the combination of continuous data with discrete data can be challenging. As Koller notes, "even representing the correct marginal distribution in a hybrid network can require space that is exponential in the size of the network" [28, p. 617]. However, in many of our real-world use cases for Bayesian networks, continuous and discrete data are mixed. An alternative to using hybrid models would be to transform continuous data into a discrete form: discretization. This is not only useful in the area of Bayesian networks but finds applications in a wide range of machine learning tasks [30]. Before continuing to the next part of this paper, we will review several discretization techniques. For a more in-depth overview of existing discretization approaches, see [14, 44].

2.4.1 Median/IQR

One of the simplest approaches of discretization is based on the median value. This discretizer would partition a set of continuous values into two groups: those smaller than the median, and those greater than or equal to the median. Both groups can then receive an arbitrary label. Since this method always creates at most two partitions (also bins or buckets), it can be used when we do not want a high model complexity. An alternative to the median could be based on the interquartile range (IQR): we partition the set in three groups, the first being the first quantile, the second being the second and third quantile, while the third contains the data of the fourth quantile. IQR can be used for sets of values where we want to separate the low from the high and have a large base of medium values.

2.4.2 Equal Width Discretization (EWD)

Equal Width Discretization, while still relatively simple, is a slightly more elegant method. Its goal for discretization is to divide the set of continuous variables into partitions that all span the same interval size. Datapoints get labeled according to the partition they belong to. The number of groups EWD has to create is left as a hyperparameter. A disadvantage of this approach is that outliers can significantly influence the way bins are created. If, for example, we have 100 data points in a range of $0 - 10$, one data point at $-1,000$ and one at $1,000$, and we want EWD with three bins, 100 datapoints would be put into the same middle partition. The two other bins would only contain a single data point. Would we have used ten bins, this would not have changed, but we would have created seven additional empty bins. This inefficiency causes much information to be lost in the discretization process.

2.4.3 Equal Frequency Discretization (EFD)

Equal Frequency Discretization is an alternative to EWD which tends to be more resilient to outliers. Its approach to discretization is to create bins, not of the same interval size, but that contain equal amounts of data points. In our previous example, rather than having many empty bins, we would create two bins that had rather large intervals, but we would have retained a lot of information in the other bins. In this thesis, EFD will be our preferred choice when we need to choose a discretization strategy. We will do so because creating categories with equal frequencies prevents the (D)BN learners from being biased due to class imbalance.

Chapter 3

Classification DBNs

In this chapter, we look at solving the classification problem for temporal datasets while maintaining explainability. We will do so by introducing the concept of Classification-Dynamic Bayesian networks (or C-DBNs), which are a subset of DBNs. Some ideas for C-DBNs have been inspired by another subset of models that can be represented as a DBN: Factorial Hidden Markov Models (FHMMs) [17]. There are a number of key differences though, which we think set C-DBNs apart as a useful group in its own right. In this chapter, we will first look at the basic architecture of C-DBNs, spend some time on formalizing them, and finally conclude with a brief overview of the differences between C-DBNs and FHMMs to illustrate the usefulness of this newly created group of models.

3.1 Architecture of C-DBNs

In this project, we have two distinct objectives: we not only want to construct a model that is capable of classifying situations over time, but we also want this model to be explainable. Either task on its own would lead to a rather straightforward way of designing the model: for the former, we optimize the predictive accuracy of the classifier, while for an explainable model we require an accurate underlying generative system. While bringing these two goals together will inevitably lead to some loss of performance at the individual level, we hope that their joint real-world practicality will outweigh the loss in performance.

To create a balance between the two, we have opted to use DBNs as a basis for our research. As shown in Section 2.3, this group of models is used for real-world situations in which both accuracy and explainability are required. We have seen before however that DBNs are not specifically made for classification, but for reasoning over an entire domain. To attain the highest performance, we want to identify a set of constraints on DBNs, which form a subset that is dedicated to classification, while retaining the core principles of accuracy and explainability.

To achieve this, we took inspiration from FHMMs which model time-series by representing all information in a number of state variables that are connected over time [17]. All these states are hidden and are the parent of a single observation. This model seems close to what we want: a single target with multiple nodes connected to it. In our case, we want to have the roles inverted however: we want multiple observations to lead to one hidden target (i.e., the classification label) that we are predicting. We will, therefore, rotate the relationships between observations and state.

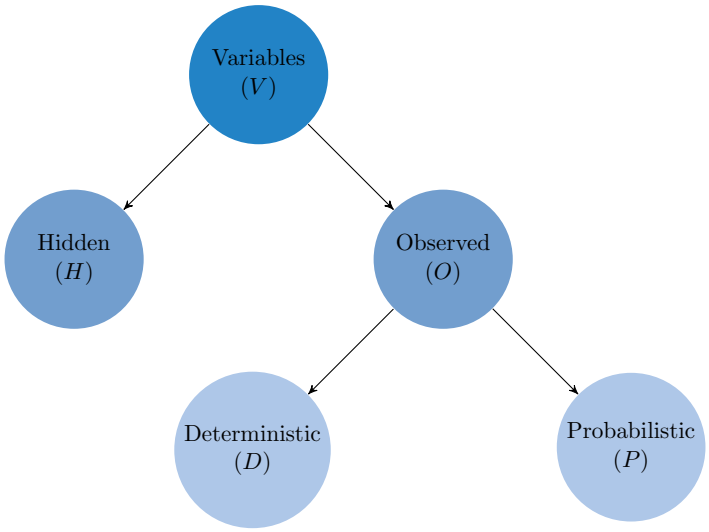


Figure 3.1: Visual overview of the types of variables that comprise a C-DBN.

Using these ideas, we specify for C-DBNs the following: we want a single hidden state (the classification target) and the possibility to define multiple observations. The observations are parent nodes of the target. In addition, while nodes can only be connected between directly adjacent timeslices in FHMMs, we will allow C-DBNs to be more expressive by allowing relationships that "jump" over timeframes. Another change we make to create the group of C-DBNs is that the classification target can be influenced both by the observations from the past (within a given range) and the observations from the present. This will allow us to react to patterns over time.

We also expand on the idea of distinguishing between the hidden state and the known observation. Not only do we set our classification target to be the hidden state, but we also split the observations into two further groups: probabilistic and deterministic ones, as shown in Figure 3.1. The former are true observations in the sense that they have to be perceived or sensed, while the deterministic ones can be computed instead. In practice, this means that the deterministic variables include features such as 'time of day' and 'month', while uncertain ones can include measurements such as 'length' and 'temperature'.

Deterministic variables are often linear and follow a strict order (e.g., months cannot "jump ahead"), which causes them to be in conflict with the frequency-based learners that Bayesian networks tend to rely on. These learners generally do not take order into account, and because of that, can make mistakes like always predicting the value that occurs most frequently in the dataset. A network that has been learned on data from January and February could for example always predict January as next month, even if the previous date is in February (as January has more days).

To avoid these problems associated with deterministic variables, we could fuse our automatically generated network with human input directly: we could represent the correct transitions in the joint probability distribution of the model with ones and zeros. However, in this project, we decided to preclude deterministic nodes from having parents and supplied the values to the model as evidence. While both approaches are equivalent, the latter can be more convenient to implement since distributions do not need to be altered.

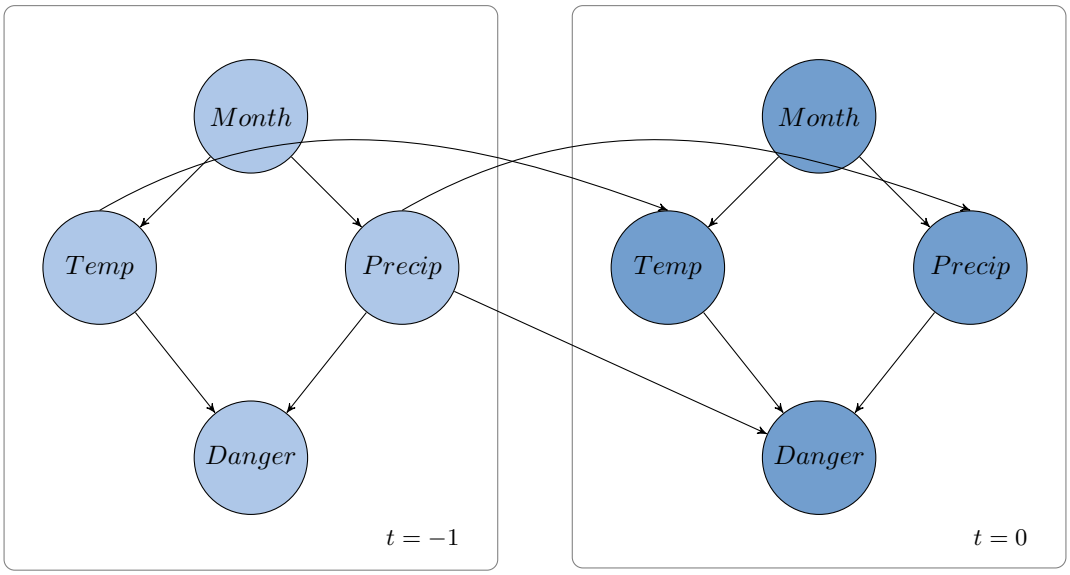


Figure 3.2: Example of the proposed C-DBN architecture with four nodes: state *Danger*, probabilistic observations *Temp* and *Precip*, and deterministic observation *Month*. Note that the two hidden nodes, $Danger^{t=-1}$ and $Danger^{t=0}$ are not connected. The deterministic node *Month* has no parents. The state in $t = 0$ is able to receive evidence from the previous layer.

3.2 Formalizing C-DBNs

We will now formalize these ideas by adding a number of properties and constraints to an existing DBN $\mathcal{G} = (V, E, \Theta)$, with vertices V, edges E, and joint probabilities Θ. We refer to the timeframe t to which a vertex v belongs by v^t. By convention, we denote the present as $t = 0$, the past as $t = n$ for $n < 0$ and the future by $t = n$ for $n > 0$ where $n \in \mathbb{Z}$. We say that vertex x is a parent of y or $x \in Pa(y)$ iff $(x, y) \in E$. For the sake of explainability and reducing model complexity, we assume stationarity, i.e. edges and probabilities do not change over time.

Using this notation we can describe, as is common for DBNs, the Markov assumption. This means we assume that variables in the past do not influence the present given the previous n timeframes. We also enforce, as usual, that time flows forward in equal increments and that the future cannot influence the past:

Constraint 3.1 (Progression of Time)**:**
There are no edges (x^i, y^j) for any combination of vertices $x, y \in V$ if the parent vertex x^i occurs in a later timeframe than the child vertex y^j: $(x^i, y^j) \notin E$ if $i > j$.

Constraint 3.2 (n^{th}-order Markov-assumption)**:**
For the n^{th}-order Markov assumption, there is no edge $(x^i, y^j) \in E$ between variables x^i and $y^j \in V$ if $j - i > n$.

For C-DBNs, we also introduce the notion of vertex types: a vertex can either be hidden or observed. We denote the set of hidden vertices in $\mathcal{G}$ by H, and the set of observed vertices by O. No vertex can be both hidden and observed: $H \cap O = \emptyset$, and all variables have to be at least one: $V = H \cup O$. Observed variables can either be Probabilistic or Deterministic. We write P and D for their corresponding sets. Again, vertices cannot be both probabilistic and deterministic: $P \cap D = \emptyset$, and all observed vertices must be either probabilistic or deterministic: $O = P \cup D$. We can now introduce the constraints on behavior based on specific groups for a C-DBN $\mathcal{G} = (V, E, \Theta)$ with $V = H \cup D \cup P$:

Constraint 3.3 (Uniqueness of the hidden variable)**:**
There is only a single hidden variable: $|H| = 1$.

Constraint 3.4 (Hidden variables have no children)**:**
If a variable x is hidden, it cannot have any children: $\forall x \in H, \nexists y \in V : x \in Pa(y)$.

Constraint 3.5 (Deterministic variables have no parents)**:**
All deterministic variables have an empty parentset: $\forall x \in D : Pa(x) = \emptyset$.

Using these constraints, a C-DBN can now be defined as follows:

Definition 3.1 (Classification-Dynamic Bayesian network (C-DBN))**:**
A *Classification-Dynamic Bayesian network* is a DBN $\mathcal{G} = (V, E, \Theta)$ with Vertices V, Edges E, and the joint probability distribution Θ, where $V = H \cup D \cup P$, and:

1. The hidden variable is unique (Constraint 3.3)
2. Hidden variables have no children (Constraint 3.4)
3. Deterministic variables have no parents (Constraint 3.5)

An example of this architecture is shown in Figure 3.2.

3.3 Comparison with (F)HMMs

Since we have taken several insights from Hidden Markov Models (and FHMMs in particular), as briefly mentioned before, we will now spend some time discussing in what meaningful way we have made alterations and discuss the expected impact of these changes. First, we introduce the concept of regular HMMs and expand that to a derivative called Factorial HMMs, which is closest to our proposed C-DBNs and can also be expressed using DBNs [32]. Then we discuss how C-DBNs have been inspired by FHMMs and look at the differences and similarities between the two.

Regular Hidden Markov Models have been around since the 1960s as a probabilistic way to study signal processing [37], but have expanded to other processes as well. To model a process, an HMM is built from five parts: a number of states that are hidden n, M distinct observation symbols per state, a state transition probability distribution A, a probability distribution over the transitions in A and an initial state distribution π [37]. Using these components, HMMs predict state transitions based on observations. If we want to predict the weather (which is the *state* of the HMM) for tomorrow for example, we could observe the current state as being rainy (an observation in M) and use the probabilistic model over the possible transitions in A to calculate the probability for all possible options, such as snowy or sunny.

Such a model is rather limited since it does not make use of potentially interesting other information. One could imagine that to predict the weather of the next day, knowing the season or the current temperature would also be relevant. Maybe we want to focus on a specific part of the weather. To do so, we can extend the regular HMM to a Factorial HMM, which still has a global state but also splits this up into smaller components [32, p. 20]. We would still use the weather as a hidden state, but also decompose it into smaller parts such as temperature, precipitation, and season. These individual components also have their transition model: they are influenced by their previous value as well as influencing the global hidden state. There is no direct interaction between the smaller components, however.

It is from this Factorial HMM that we have taken inspiration to create C-DBNs. They share the idea that the global state can be split into smaller components, which influence the global state. In our model, however, we wanted to go a step further and lift the constraint that the individual variables cannot influence another. Lifting the constraint provides a more explainable view of the process which is being modeled, as correlations are directly visible from the relationships instead of going through a state first.

Like Factorial HMMs but unlike regular HMMs, C-DBNs also do not allow the previous global state to influence the current state. We have chosen this approach for two reasons. First, we want to force the model to re-evaluate the current state based on the current evidence alone, because it makes it more apparent why a particular prediction has been made. We also worry that relying too heavily on the current prediction decreases the transparency of the model. The second argument is more pragmatic: since we allow individual components to be connected, we are potentially creating conditional probabilities that involve many more variables. When we want to machine-learn models, as we will discuss in a later chapter, this means more room for overfitting. When we allow global states to be connected, we will often learn models in which the next state is predicted by replicating the current state. The resulting network is neither very explainable nor very accurate. To prevent this from happening, we leave global states without a direct connection.

Another difference between the proposed new architecture and HMMs in general, is that they work with other types of data. HMMs generally try to model processes (by using state transitions for example), meaning changes in a state. The proposed C-DBNs, on the other hand, are meant to work with temporal datasets. While temporal data usually also describes a process, it is often measured using discrete and static time-intervals. C-DBNs are therefore less focused on change and more on predicting a given situation or context correctly.

To conclude, while C-DBNs take some ideas from (F)HMMs, such as the decomposition of the state into individual components with their transitional probability distributions over time and the isolation of global states, we have also made several changes that set it apart. While our focus is more on temporal datasets with an underlying process which we want to describe, HMMs look at change points (i.e., points at which the distribution of a variable changes [35]) in the process and describe it more directly. Both are valid approaches but have different applications. Where HMMs can be used in processes that often change, like for speech recognition [37], C-DBNs are more focused on datasets with fewer change points, such as classification over time.

Now we have seen how the group of C-DBN models fundamentally differ from the related FHMMs and how they can potentially perform better in temporal classification tasks, we will make them convenient to use in practice. For this, we should be able to learn C-DBNs automatically from data. The next chapter will continue on that topic.

Chapter 4

Learning C-DBNs from Data

While in the past we were limited to creating Bayesian networks by hand by questioning domain experts, we are now able to obtain them by machine learning if we have enough data. Learning C-DBNs, the architecture we discussed in the previous chapter, requires several adaptations to existing BN-learning methods. Since these adaptations generally overlap with learning DBNs, we will first focus on learning DBNs in general to make this method as widely applicable as possible. We will then describe the various constraints that are required for learning C-DBNs in particular. We do not use existing libraries for learning DBNs since they are not widely (and openly) available for modern data science frameworks and would not allow us to make complexity optimizations that are specific to C-DBNs. There also does not seem to be a standard for learning DBNs, so in this chapter, we aim to provide a recipe that future work can build on.

We will first introduce the concept of shifting, which transforms a temporal dataset to a regular one. Then, we will look at the constraints which are required for learning valid structures. This will be followed up by an overview of how the output of a BN-learner can be transformed into a DBN structure. We will conclude the overview by discussing how the transformed structure can be unrolled over time. Throughout the chapter, we will, in addition to the theoretical notation, describe how the methods can be implemented. This will mostly be done in a Python-style pseudo-code notation. At the end of the chapter, we will describe the implementation differences that are specific to C-DBNs.

4.1 Existing work

Machine learning the structure of a Bayesian network is a computationally intensive task. The duplication of nodes that we need for creating a dynamic network magnifies this problem even further. In order to still be able to learn in non-trivial contexts within a reasonable amount of time, it is therefore imperative that we make use of an efficient implementation for structure learning dynamic Bayesian networks.

For regular Bayesian networks, much research has been done for efficient learning [7, 13, 40]. Notable is James Cussen's implementation of GOBNILP [12], a program that performs structure learning on data using integer linear programming. As recreating such a system for dynamic Bayesian networks would be a rather time-intensive task, we prefer to make use of existing work.

To be able to use existing BN toolkits for learning DBNs, we use Pavlović et al.'s [34] insight that

the problem of learning a dynamic Bayesian network can be rewritten as a Bayesian network learning task. We structure the process of rewriting as three steps: first we augment our data set to allow regular learners to use temporal data, then we add several constraints to ensure we obtain a DBN structure, and finally, we transform the resulting network back to a DBN. This works in practice for all DBNs, not just C-DBNs. The additional assumptions which C-DBNs make can be added as constraints to the structure learner.

4.2 Shifting data

For learning from a dataset $\mathcal{D}$ consisting of a feature set X with n elements in each vector, the first step involves augmenting the dataset by adding a feature vector v_x^t for each existing feature vector $x \in X$ and for each time frame $t \in [-l, 0]$, in which the constant l is the maximum number of time steps we want to look back. The new vector v_x^t for x at t is constructed from the original vector by shifting the values by t:

$$v_x^t = (x_{l+(1+t)}, x_{l+(2+t)} \cdots, x_{n-(l-t)})^\top \tag{4.1}$$

Note that for $t < l$, we are dropping the last t values of a vector. We do this because, for shifting, we need to look back in time, and for the first l elements we miss this information about the past. When we augment the dataset, this means that we use the last l elements for shifting up, and need to compensate on earlier timeframes by removing the last elements. This will become more clear in an example later on. Our augmented dataset can now be described as follows:

$$\mathcal{D}_{\text{augmented}} = \{v_x^t \mid x \in X, t \in [-l, 0]\} \tag{4.2}$$

This process is shown programmatically in Listing 4.1. We now illustrate it by means of an example. Table 4.1 shows our original dataset $\mathcal{D}$ with $n = 3$ elements for two variables, v_α and v_β. To augment our dataset so it can be used for learning a dynamic model, we will shift this dataset with one timeframe ($l = 1$). To obtain augmented dataset $\mathcal{D}'$, we use Equation 4.2:

$$\begin{aligned}
\mathcal{D}' &= \{v_x^t \mid x \in X, t \in [-1, 0]\} \\
&= \{v_x^t \mid x \in \{v_\alpha, v_\beta\}, t \in [-1, 0]\} \\
&= \{v_\alpha^{t=-1}, v_\alpha^{t=0}, v_\beta^{t=-1}, v_\beta^{t=0}\}
\end{aligned}$$

We can now create the augmented variables in $\mathcal{D}'$ by using Equation 4.1. We start with $v_\alpha^{t=-1}$:

$$\begin{aligned}
v_\alpha^{t=-1} &= (x_{1+(1-1)}, x_{1+(2-1)})^\top \\
&= (x_1, x_2)^\top \\
&= (9, 11)^\top
\end{aligned}$$

For $v_\alpha^{t=0}$ we can use the same equation:

$$\begin{aligned} v_\alpha^{t=0} &= (x_{1+(1-0)}, x_{1+(2-0)})^\top \\ &= (x_2, x_3)^\top \\ &= (11, 10)^\top \end{aligned}$$

Here we see that were are reducing the size of the dataset by one row, since $l = 1$. The case for v_β works analogously. By adding the four augmented vectors to a new set, we have created our augmented dataset $\mathcal{D}'$. The final result is shown in Table 4.2.

v_α	v_β
9	3
11	3
10	3

Table 4.1: Original dataset $\mathcal{D}$

v_α^t	v_β^t	v_α^{t-1}	v_β^{t-1}
11	3	9	3
10	3	11	3

Table 4.2: Augmented Dataset $\mathcal{D}'$ with $t = 1$

```python
def create_lookback_dataset(df, look_back=1):
    transformed_set = df.copy()

    # Add shifted columns
    for delta in range(1, look_back + 1):
        for column in df.columns:
            transformed_set[(column, delta)] = transformed_set[column].shift(delta)

    # Remove top rows that contain empty cells due to shifting
    transformed_set = transformed_set[look_back:]
    return transformed_set
```

Listing 4.1: Pseudocode for shifting dataframes

This process of shifting allows regular BN learners to make connections between time frames. Note that in this example, we only wanted to learn from the previous time frame, but the same could be applied for any $l < n$.

4.3 Constraining parent sets

If we applied any structure learning algorithm directly to the augmented dataset created in the previous subsection, it would most likely learn an invalid DBN: it would be able to create an edge from the future to a vertex in the present. We therefore need to add the constraint that a vertex cannot be the child to any vertex in a time frame later than itself. We also enforced this for C-DBNs in Constraint 3.1.

While a naive approach in which this constraint is checked while the graph is being generated would produce a correct result, it does not take advantage of the opportunity for reducing computational complexity that this constraint creates. Since a vertex cannot access any vertex in a later time frame, we can skip all calculations that would be required for using evidence from the future in the past, thereby significantly lowering the computational burden created by the increase in nodes that the dataset transformation entails.

For the purpose of this project, we limit our experiments to generating parent sets for first-order Markov networks, i.e., networks with a current layer $t = 0$, and one look-back time frame

$t = -1$. We adapted GOBNILP's [12] regular parent set generation system and split it up so it only generates the scores between a single vertex and the rest of the network, so we can influence the process of structure learning to make it more efficient. Then we use this to create a new function, which calculates the scores for all vertices in $t = 0$ and leaves all vertices within $t = -1$ without parents. This significantly reduces the complexity of structure learning by removing edges from being computed that will be dropped in the transformation. While we only look at first-order Markov networks here, this strategy could be generalized to a system in which multiple timeframes are investigated by only computing scores in the same frame and before it.

The result of generating these parent sets to learn a network is that nodes within $t = 0$ have connections among themselves as well as from the past. Vertices in $t = -1$ have no connections at all. In the next subsection, we will transform the learned structure into a proper DBN.

4.4 Transforming the graph

While the result from the previous section has the right specifications of a DBN structure, it lacks a number of edges. Particularly, because we assume stationarity as noted in Section 3.2, all vertices that are connected in $t = 0$ with other vertices in $t = 0$ should be connected in all other time frames as well. More formally, for an n^{th}-order Markov assumption, we require that:

$$E_{\text{transformed}} = E_{\text{learned}} \cup \{(x^{t=i}, y^{t=i}) \mid (x^{t=0}, y^{t=0}) \in E_{\text{learned}}, i \in [-n, 0\rangle\} \tag{4.3}$$

We can now use the reconstructed structure defined by the transformed edges $E_{\text{transformed}}$ and the augmented dataset $\mathcal{D}_{\text{augmented}}$ to learn a transformed full joint probability distribution $\Theta_{\text{transformed}}$ using any parameter learning tool, as if it were a regular Bayesian network. In this project, we use PGMPY's Bayesian Estimator [4] to find the necessary conditional probabilities.

4.5 Building the unrolling framework

The resulting network allows us to either predict one step into the future or query the probability of variables given an incomplete present or past. If we want to expand on this to an arbitrary number of time steps, we need to unroll the network. This encompasses duplicating nodes for each time frame and connecting them properly.

Duplicating nodes involves using a framework that splits DBNs conceptually into two distinct parts: a **base** layer and a transition layer [32]. For an n^{th}-order Markov network the *base* layer consists of n replications of all nodes V: one for each time frame: $\mathcal{G}_{\text{base}} = (V_{\text{base}}, E_{\text{base}}, \Theta_{\text{base}})$ with $V_{\text{base}} = \{x^{t=i} \mid x \in X, i \in [-n, 0]\}$ and $E_{\text{base}} = E_{\text{transformed}}$. This corresponds directly to the structure that we have learned. We can now also use $\mathcal{D}_{\text{augmented}}$ to learn the joint coditional probabilities Θ_{base} using a parameter learning system.

For the second part, or the **transition** layer, we want to create a method of easily attaching layers. To do so, we create a single timeframe that contains all the edges within the frame, and set a specification for how to connect it to other frames. We can then specify the transition model as $\mathcal{G}^{t=i}_{\text{trans}} = (V^{t=i}_{\text{trans}}, E^{t=i}_{\text{trans}}, \Theta^{t=i}_{\text{trans}})$ with $V^{t=i}_{\text{trans}} = \{x^{t=i} \mid x \in X\}$. If we have a first-order model and therefore a base layer of two time frames ($t \in [-1, 0]$), we can describe the set of edges for our transition layer as follows:

$$E_{\text{trans}}^{t=i} = \{(x^{t=i}, y^{t=i}) \mid (x^{t=0}, y^{t=0}) \in E_{\text{base}}\} \cup \{(x^{t=i}, y^{t=i-1}) \mid (x^{t=0}, y^{t=-1}) \in E_{\text{learned}}\} \tag{4.4}$$

In the first of the two sets that we create for generating $E_{\text{trans}}^{t=i}$, we copy over the edges from the base layer to this transition layer so it operates using the same underlying model. In the second part we specify how to connect the unrolling layer to the exiting network by using the edges that have been learned before. We can also generalize this function to an n^{th}-order Markov-assumption by iterating over all timeframes in the history of the assumption:

$$E_{\text{trans}}^{t=i} = \{(x^{t=i}, y^{t=i}) \mid (x^{t=0}, y^{t=0}) \in E_{\text{base}}\} \cup \{(x^{t=i}, y^{t=i-j}) \mid (x^{t=0}, y^{t=-j}) \in E_{\text{learned}},\ j \in [0, n]\} \tag{4.5}$$

Creating the set of joint probability distributions is analogous if we assume that we unfold to the right:

$$\Theta_{\text{trans}}^{t=i} = \{P(x^{t=i} | \text{Pa}(x^{t=i})) \mid x^{t=i} \in V_{\text{trans}}\} \tag{4.6}$$

with

$$P(x^{t=i} | \text{Pa}(x^{t=i})) = P(x^{t=0} | \text{Pa}(x^{t=0})) \in \Theta_{\text{transformed}} \tag{4.7}$$

in which $Pa(x)$ refers to the parents of node x as before.

We now have all the necessary ingredients for unrolling the DBN. As quickly mentioned before, there are two ways to do this. We can add a time frame on the left-hand side of the existing network, thereby conceptually creating a node further back into the past, or we can attach it to the right-hand side, which depicts the future. While the two are conceptually different, it does not matter for the behavior of the network. Since adding to the left side would require shifting probability distributions left (as they are the only nodes in the network that do not receive information from a previous layer), we prefer adding time steps to the right instead.

To add a new time frame at $t = i$ to the network $\mathcal{G} = (V, E, \Theta)$, we simply have to modify the existing set of vertices, edges, and probability distributions. This encompasses extending the set of vertices with an extra copy for each feature at i: $V' = V \cup V_{\text{trans}}^{\text{t=i}}$, connecting the new nodes to the existing network using the transition edges: $E' = E \cup E_{\text{trans}}^{t=i}$, and updating the full joint probability distribution accordingly: $\Theta' = \Theta \cup \Theta_{\text{trans}}^{t=i}$. The resulting network $\mathcal{G}' = (V', E', \Theta')$ is now a Dynamic Bayesian network for temporal dataset $\mathcal{D}$.

Listing 4.2 shows how the steps from the previous section and this section can be combined into an implementation. As in the theoretical description, we learn a structure from a shifted dataset with some additional constraints using existing tools and then extract the base from that. We then unroll the network by separating the transition layer from the base and applying that t times to the base (where t is the number of steps we want to predict into the future). After that, we also copy over the conditional probabilities that have been learned using existing toolboxes on the base model to the new layers, thereby creating a full DBN.

```
#Learns a DBN and unrolls it to a specified size
def create_unrolled_dbn(data, t_unroll, markov=1):
    ds = create_lookback_dataset(data, markov)
    lookback_model = get_structure(ds)

    dbn_base = create_base_structure(lookback_model)
    dbn_base.fit(ds)

    if t_unroll < markov:
        return lookback_model

    unroll_model = unroll_network(lookback_model, t_unroll)
    unroll_cpds(lookback_model, unroll_model, t_unroll, dbn_base)

    return unroll_model


# Unrolls a Markov-1 DBN to an arbitrary number of timeframes
def unroll_network(G, timeframes):
    unrolled_network = BayesianModel()
    base_variables = [x for x in G.nodes.keys() if get_tf(x) == 0]

    # Add nodes for all time frames
    for base_variable in base_variables:
        unrolled_network.add_node(base_variable)
        for t in range(1, timeframes + 1):
            unrolled_network.add_node((base_variable, t))

    # Add edges
    for x, y in G.edges:
        if x in base_variables and y in base_variables:
            unrolled_network.add_edge(x, y)
            for t in range(1, timeframes + 1):
                unrolled_network.add_edge(switch_tf(x, t), switch_tf(y, t))
        if y in base_variables and x not in base_variables:
            unrolled_network.add_edge(x, y)
            for t in range(1, timeframes):
                unrolled_network.add_edge(switch_tf(x, t + 1),
                                          switch_tf(y, t))

    return unrolled_network

#Unrolls the cpds of a Bayesian Model
def unroll_cpds(lookback_model, unroll_model, t_unroll, dbn_base):
    lbm_cpds = dict(zip([x.variable for x in lookback_model.get_cpds()],
        lookback_model.get_cpds()))
    lbm_cpds_delta = {k: v for k, v in lbm_cpds.items() if get_tf(k) == 0}

    for node in unroll_model.nodes.keys():
        if get_tf(node) == 0:  # first node
            unroll_model.add_cpds(lbm_cpds_delta[node])
        elif get_tf(node) == t_unroll:  # final node
            cpd = switch_cpd_timeframe(dbn_base.get_cpds(get_base(node)), t_unroll)
            unroll_model.add_cpds(cpd)
        else:  # everything in between
            unroll_model.add_cpds(
                switch_cpd_timeframe(lbm_cpds_delta[switch_tf(node, 0)],
                get_tf(node)))

# Gets the base structure of a DBN
def create_base_structure(lookback_model):
```

```
    dbn_base = BayesianModel([edge for edge in lookback_model.edges if
        get_tf(edge[0]) == 0 and get_tf(edge[1]) == 0])
    dbn_base.add_nodes_from([node for node in lookback_model.nodes
        if get_tf(node) == 0])
    return dbn_base
```

Listing 4.2: Pseudocode for unrolling the resulting BN into a DBN

4.6 C-DBN specific adaptations

In the previous sections, we have described how DBNs can be learned using regular BN learning tools. This has made it possible to learn a wide range of probabilistic temporal models. However, for this project, we want to look at C-DBNs in particular.

Adapting a DBN learning method to learn C-DBN mainly consists of adding the constraints described in Section 3.2. The exact implementation of the constraints depends on the toolkit that has been used, but here we assume that we have GOBNILP-like abilities to create constraints: we can forbid or enforce edges in the DAG. We generate the constraints by first dividing the variables up into the three groups we have mentioned before: hidden variables H, deterministic variables D and probabilistic variables P such that $V = H \cup D \cup P$. We denote the rule that node x is the child of node y as $x \leftarrow y$, and the constraint that x cannot be the child of y as $\neg x \leftarrow y$ since this closely mirrors GOBNILP's expected input format. We can now generate all required constraints for a given H, D, P as follows:

$$
\begin{aligned}
C_{\text{HiddenVariables}} &= \{\neg x \leftarrow y | (x, y) \in V \times H\} \\
C_{\text{DeterministicVariables}} &= \{\neg x \leftarrow y | (x, y) \in D \times V\} \\
C_{\text{C-DBN}} &= C_{\text{HiddenVariables}} \cup C_{\text{DeterministicVariables}}
\end{aligned}
$$

On a technical note, Gobnilp enforces constraints after having generated parent sets. To speed up this process, creating $C_{\text{HiddenVariables}}$ is equivalent to not generating any parent sets for D. This can be used to reduce complexity, especially in networks of a higher Markov-order.

Now we have specified how DBNs, and C-DBNS in particular, can be learned using existing toolkits, we are ready to use this novel approach in practice. While we do have everything required in theory, there is one more hurdle to completely automatic learning: the standard machine learning task of discretization. If we want to create a fully automatic way of learning temporal models from data, and we want to use BNs as underlying models which are typically not very capable of learning from a mixture of continuous and discrete data, we should also have a solid discretization process. While selecting a generic discretization method before the learning process, we often do not know beforehand which one will be appropriate to the data or perform well. To avoid this possible inefficiency, we want to integrate discretization into the learning process. In the next section, we will describe a way of taking the first steps to a fully automated and integrated approach, including both discretization and learning.

Chapter 5

Full Automatic Discretization for C-DBNs

5.1 The need for automatic discretization

Discretization is often done at the beginning of a machine learning job, in which the user tries to turn continuous data into categorical data while reducing the amount of information lost in the process as much as possible. In the case of DBNs, this is a vital step as these networks only have limited capacity to handle datasets comprised of both discrete and continuous data. It is not always apparent at first which kind of discretization fits well with the data, so the user can either try to match the existing distribution as closely as possible (e.g., visually by using histograms) or by checking the effect of various discretization strategies on the training accuracy.

While manually finding a discretization approach might be a feasible approach for some small data sets with similar distributions across variables, finding the optimal discretization method for larger datasets with a multitude of variables with their distinct distributions will quickly become too time-consuming. This holds especially if we are less interested in adhering to the original distribution, but would rather use a strategy that aims to optimize the joint predictive power of the various discretized variables and their synergistic effects. We would, therefore, prefer to have this important task taken away from us and choose an optimal discretization strategy automatically.

5.2 Towards full automatic discretization

To continue in the spirit of this project, i.e., solving problems by creating transformations to existing problem solvers, we want to reduce the problem of choosing a discretization strategy to one of structure learning. Doing so allows us to move more tasks away from the user onto existing toolboxes, thereby enhancing real-world usability.

In this chapter, we will first consider a simple Dynamic Bayesian network with only the current time frame (0^{th}-order Markov assumption), which is equivalent to a regular Bayesian network. We create a network with two correlated variables, one of which is continuous and the discrete target. From there, we will build further towards using multiple correlated continuous variables. Then we show how to expand this approach to first-order Dynamic Bayesian networks. Finally,

we will address how this method can be applied to any n^{th}-order C-DBN. Every time, we will first introduce the theory behind the transformation, then show the transformation in action using an example. In the last part of this chapter, we will experiment with the proposed theory and analyze the results.

5.3 Discretizing a single continuous variable

Theory

We start with a 0^{th}-order DBN with one probabilistic variable and one target, which equates to a regular Bayesian network. We want to create this network from a dataset $\mathcal{D}$ which consists of two columns (represented as two vectors): a vector of real values $v \in \mathbb{R}^n$ and one of the discrete target values $y \in \mathbb{N}^n$. Since we want to learn from a fully discrete BN, we will have to discretize column v. We will transform this into a structure learning problem using multiple binary discretization nodes.

A binary discretization node d for v splits a set of numbers in two by using a single boundary: $r \in \mathbb{R}$. Using this discretization node we can represent information about v in a discrete (though by itself limited) way. We transform a vector of values according to this boundary by creating a value that is `False` (or zero) if the original value of the element is below the boundary and `True` (or one) otherwise. The feature vector for discretization node d_v^r can therefore be constructed from v as follows:

$$d_v^r(i) = \begin{cases} 0 & \text{if } v_i < r \\ 1 & \text{otherwise} \end{cases}$$

Since we want to consider all possible information which we can extract from v using this concept of discretization nodes, we will create a discretization node for each possible boundary between the values in v. To do so, we first sort the vector and remove duplicate values to avoid superfluous nodes. Placing boundaries can be done in multiple ways, but we have chosen for the arithmetic mean. By doing this, we obtain a set of discretizers D_v for v:

$$D_v = \{d_v^r \mid r = \frac{v_i + v_{i+1}}{2}, i \in [0, n\rangle\}$$

We can now create an augmented dataset $\mathcal{D}'$ which contains the data for $n-1$ discretization nodes for v, and a single node for the already discrete y. By feeding this augmented dataset to a structure learner, we will obtain relationships between the most important discretization nodes and the target. Since we are building towards C-DBNs in which the target node has no children, we can say that a discretization node d_v^r is **selected** if there is an edge from it to the target node y:

Definition 5.1 (Selected Node)**:**
Discretization node d_v^r is **selected** given a learned structure $\bar{G} = (\bar{V}, \bar{E})$ and target y if $(d_v^r, y) \in \bar{E}$

We can then specify the set of selected nodes S_v for variable v as:

Definition 5.2 (Selection Set)**:**

The **selection set** S_v for v given a learned structure $\bar{G} = (\bar{V}, \bar{E})$ and target y consists of all discretization nodes for v that are selected: $S_v = \{d_v^r \mid d_v^r \in D_v \text{ if } d_v^r \text{ is selected in } \bar{G}\}$

The boundaries r corresponding to the discretization nodes in S_v form the *discretization strategy*:

Definition 5.3 (Discretization Strategy)**:**

The **discretization strategy** R for variable v consists of the set of boundaries specified by its selection set $R_v = \{r \mid d_v^r \in D_v\}$

Using the discretization strategy R, we can now partition variable v into $|R| + 1$ groups by creating a cutting point at each boundary. By applying this discretization strategy to the dataset, we can now continue using our regular BN-learning toolboxes for structure learning and parameter learning.

We think that this approach could work because it lets the structure learner pick the nodes which help it create the most likely graph given the data. If a variable is unlikely to influence the rest of the variables, it does not get connected. If a variable is connected to something else than the classification variable, it does not contribute to the final discretization. Next, we will look at an example to illustrate how the proposed model of discretization works.

Example

To illustrate this process we take a dataset $\mathcal{D}$ consisting of a continuous variable v with values between zero and one, and a binary target y. This example has five rows and is depicted in Table 5.1. From this dataset, we create four discretization nodes by taking the boundaries between the sorted values:

$$\begin{aligned} D_v &= \{d_v^{0.5\cdot(v_0+v_1)}, \ldots, d_v^{0.5\cdot(v_3+v_4)}\} \\ &= \{d_v^{0.5\cdot(0.024+0.043)}, \ldots, d_v^{0.5\cdot(0.881+0.973)}\} \\ &= \{d_v^{0.034}, d_v^{0.306}, d_v^{0.724}, d_v^{0.927}\} \end{aligned}$$

v	y
0.024	0
0.043	0
0.568	0
0.881	1
0.973	1

Table 5.1: Dataset $\mathcal{D}$

$d_v^{0.034}$	$d_v^{0.306}$	$d_v^{0.724}$	$d_v^{0.927}$	y
0	0	0	0	0
1	0	0	0	0
1	1	0	0	0
1	1	1	0	1
1	1	1	1	1

Table 5.2: Dataset $\mathcal{D}'$

v'	y
0	0
0	0
0	0
1	1
1	1

Table 5.3: Discretized $\mathcal{D}$

We then create an augmented dataset $\mathcal{D}'$ from the new nodes by using their boundaries for classification. This is shown in Table 5.2. After that, we use a structure learner with the BDeu score (in this case GOBNILP) to find the selection set. The resulting learned network is shown in Figure 5.1. As shown in the figure, only discretization node $d_v^{0.724}$ is selected (i.e., it is the parent of the target y). We will, therefore, discretize vector v based on the discretization strategy that has a single boundary at $r = 0.724$. This creates v', which is shown in Table 5.3. If we had multiple selected nodes, we would have discretized into multiple categories, as we will

see later. Using the new, discretized dataset, we can now restart our structure learning process and learn that there should be an edge between v' and y. We can then continue learning the parameters using a regular BN-toolbox. In this way, we have learned a discretization strategy as well as a complete network by using existing methods.

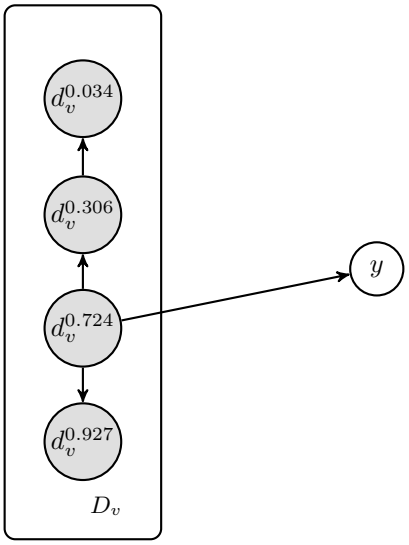


Figure 5.1: Visual representation of the learned discretization network.

5.4 Discretizing multiple continuous variables

Theory

Now we have presented the case for creating a Bayesian network with automatic discretization from a single continuous variable and a single target, we will move on to datasets with multiple continuous variables. First, let us assume that we have a dataset $\mathcal{D}$ which consists of m columns (represented again as vectors): $m-1$ containing real values and one target vector y consisting of discrete values. All pairs of vectors can be correlated. Our goal this time is to create an optimal discretization strategy for all input features given y as the target. We assume as before that the values in the vectors are unique and that they can be sorted.

Using this information, we can again construct discretization nodes for all boundaries for all continuous variables in the dataset and learn a structure $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ as before. As with C-DBNs, we do not allow the classification variable to have any children to maintain consistency. We also will not allow discretization nodes for the same variable to be connected to another. We make this choice because we are interested in the effect of the discretization nodes on the target, and not on each other. Adding the constraint therefore reduces complexity without impacting our goal.

Now we have to choose which discretization nodes should determine our discretization strategy. Finding which nodes influence our target is equal to the concept of finding the Markov Blanket (MB) for y. The MB includes the parents of y, its descendants, and the parents of its descendants [29, p. 32]. These nodes provide all the relevant information for y since all other nodes in the network are conditionally independent of y given the MB [29, p. 284]. Since our target variable does, by the constraints we have made, not have any children, our discretization strategy consists of selecting all nodes that are the parent of y as before.

While this solves the problem of knowing which nodes to select, we should also study how adding more columns influences our problem space. Where before we required $n-1$ nodes for discretization with n being the number of distinct rows, we now need $(m-1)(n-1)$ additional variables: $n-1$ for all $m-1$ continuous variables. If we stick to binary nodes for discretization, this means that we will have $2^{(m-1)(n-1)}$ possible assignments for a data point. Since the size of the dataset that we require for accurate learning grows with the number of possible assignments, this means the dataset we need grows exponentially with the number of both the original variables and the number of entries. While this is not a problem in the case of a single continuous variable as adding a new data point will not lead to multiple new combinations, this is, in fact, a problem for the approach with multiple continuous variables. Every new unique datapoint we add increases our demand for data with combinations with the other variables, or we risk losing accuracy. Adding more data points would by itself, however, also increase our need for data, creating an endless loop.

We could potentially mitigate these problems by limiting the precision of our real values, which can be seen as a limited form of unsupervised discretization. For this project, we will restrict ourselves to networks with a small enough number of continuous variables for the direct approach to be feasible.

Example

To illustrate how our theory works, we will now look at a rather simple example. We have created a dataset $\mathcal{D}$ consisting of two continuous columns v, w and binary target column y. Vector v is created by uniformly sampling twelve real values between zero and one, while w is created from v by multiplying it by 1.1 and adding some normally distributed noise ($\mu = 0$, $\sigma = 0.05$). The target y has been constructed from both v and w as follows:

$$y_i(v, w) = \begin{cases} 1 & \text{if } 0.5 < v_i < 0.85 \text{ and } 0.5 < w_i < 0.85 \\ 0 & \text{otherwise} \end{cases}$$

A generated example is shown in Table 5.4 and Figure 5.2. We create $n-1 = 11$ discretization nodes for both v and w in the same way as before. This results in a discretization dataset $\mathcal{D}'$ of $2(n-1)+1 = 23$ columns. The result is shown in Table 5.5. We now add the two constraints: no discretization node can be connected to another discretization node for the same variable, and y cannot have any children. From this, we can use existing toolboxes to learn the structure of the network. The result is shown as a graph in Figure 5.3.

We can now see from the learned edges that discretization nodes $d_v^{0.5}, d_v^{0.75}, d_w^{0.57}, d_w^{0.81}$ are selected. We will therefore split v on 0.5 and 0.75, and w on 0.57 and 0.81 to obtain discretized dataset $\mathcal{D}'$, shown in Table 5.6. These boundaries are a good approximation of the decision boundaries we have set up for the data before. We can now learn a BN from $\mathcal{D}'$ using the regular structure learning tools. A possible result of this is shown in Figure 5.4.

v	w	y
0.080741	0.029177	0.0
0.187721	0.196250	0.0
0.206719	0.209450	0.0
0.221993	0.274366	0.0
0.296801	0.243241	0.0
0.488411	0.502243	0.0
0.518418	0.627829	1.0
0.611744	0.765785	1.0
0.738440	0.736725	1.0
0.765908	0.874741	0.0
0.870732	0.908775	0.0
0.918611	0.967629	0.0

Table 5.4: Example dataset with continuous variables v, w and discrete target y

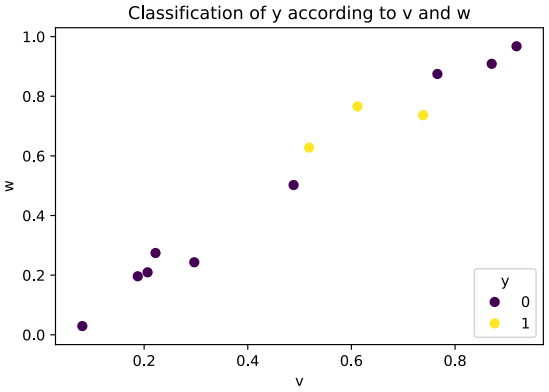


Figure 5.2: Scatter plot of how y depends on v, w

v(0.13)	v(0.2)	v(0.21)	v(0.26)	v(0.39)	v(0.5)	v(0.57)	v(0.68)	v(0.75)	v(0.82)	v(0.89)	w(0.11)	w(0.2)	w(0.24)	w(0.26)	w(0.37)	w(0.57)	w(0.7)	w(0.75)	w(0.81)	w(0.89)	w(0.94)	y
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0.0
0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0.0
0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0.0
0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0.0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0.0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	0.0
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1.0
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1.0
0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1.0
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0.0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0.0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.0

Table 5.5: Example dataset with continuous variables v and w and discrete target y

v	w	y
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	1	0
1	1	1
1	2	1
1	1	1
2	2	0
2	2	0
2	2	0

Table 5.6: Resulting $\mathcal{D}'$ of automatic discretization of $\mathcal{D}$

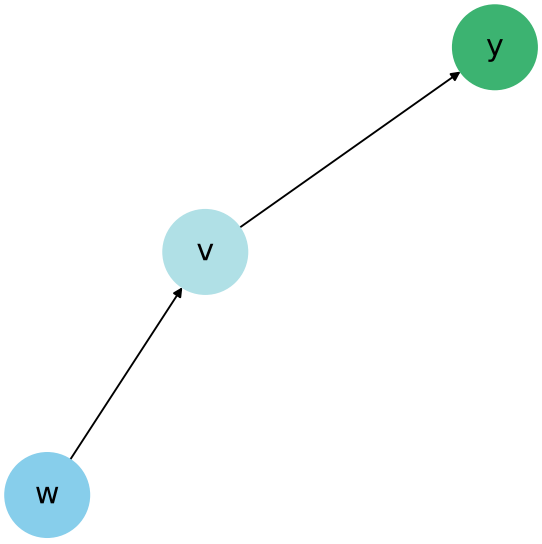


Figure 5.4: Learned network based on discretized dataset $\mathcal{D}'$

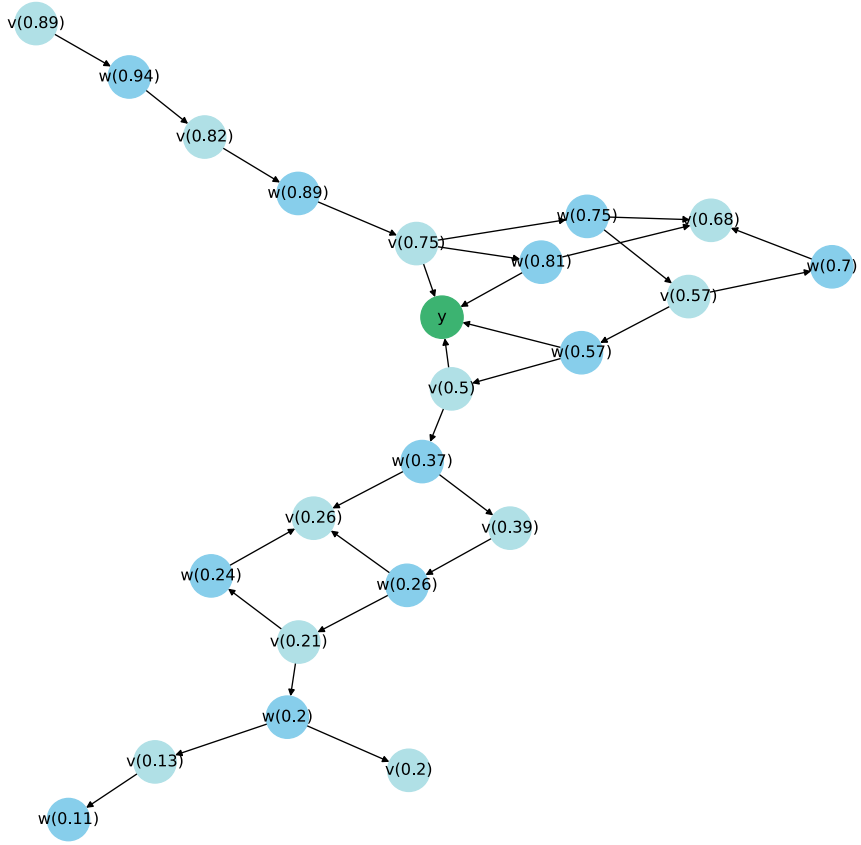


Figure 5.3: Visual representation of the learned discretization network with continuous variables v and w and discrete target y.

5.5 1^{st}-order C-DBN with multiple continuous variables

Theory

As we have seen in Chapter 3, we can transform the problem of learning C-DBN structures to be an instance of the BN-structure learning problem. This means that, for 1^{st}-order Classification Dynamic Bayesian networks we have a copy for all variables in $t = 0$ and $t = -1$. Since we have specified automatic discretization in terms of structure learning, we can combine these two transformations and use them to learn a discretization strategy for DBNs automatically. To accomplish this, we first shift the dataset, as in Section 4.2 and add the regular constraints for DBNs and C-DBNs, mentioned in Section 3.2.

While this lays down the foundation for learning our discretization strategy, we have one more challenge to deal with. We now have more than one observation: we have both a $y^{t=0}$ and a y^{-1}. The case can be made that if a boundary helps predicting y node in time frame $t = -1$, it should also be included in the discretization strategy for t since it adds information to the classification process. We would argue however that since in our case, all variables in $t = -1$ are always given and it is our objective to predict $y^{t=0}$ this would add superfluous information that makes the model needlessly complex. Remember that, if the edge would be beneficial for predicting $y^{t=0}$ the structure learner has the power to connect them. Therefore, we do not lose information by excluding nodes connected to $t = -1$ and use our way of selecting boundaries as before.

Although we have now formulated a rather straightforward process of learning a discretization strategy for DBNs, we should address the algorithmic complexity of this solution. In the previous section, we have already seen that we create $2^{(m-1)(n-1)}$ possible combinations, with m being the number of continuous variables and n the number of rows. By duplicating each vector in the shifting process for creating a DBN learner, we expand this even further to $2^{2\cdot(m-1)(n-1)}$ possibilities, quickly shrinking the set of problems in which this technique can be applied even further. As before, we can mitigate this problem somewhat by exploiting the constraints on DBNs to limit the search space of the parent set generation, but the speed at which the network grows will quickly make that approach untenable as well. The user would then need to rely on even lower precision or start sampling or approximation to be capable of using this approach for anything but very small datasets.

Note that even though we only look at the classification node in $t = 0$ for discretization, this does not mean that we can learn based on $t = 0$ alone. We can expect very different graph structures when the additional nodes that represent time are included. The graph from which the structure learner would determine a discretization strategy would otherwise be very different from the one the final model is learned from, and interaction effects could be ignored. It is therefore not possible to prune the model to $t = 0$.

Example

For this example we simulate a time series using the sine function as follows:

- $v(i) = \sin(i * 360/T) + \mathcal{N}(0.2, 0.2)$
- $w(i) = v(i-1) + \mathcal{N}(0.4, 0.1)$

v	w	y
1.557726	0.654749	0.0
2.153327	1.895026	1.0
0.353274	2.548946	1.0
0.827301	0.705552	0.0
2.010548	1.095915	1.0
0.263023	2.499010	1.0

Table 5.7: Generated dataset $\mathcal{D}$

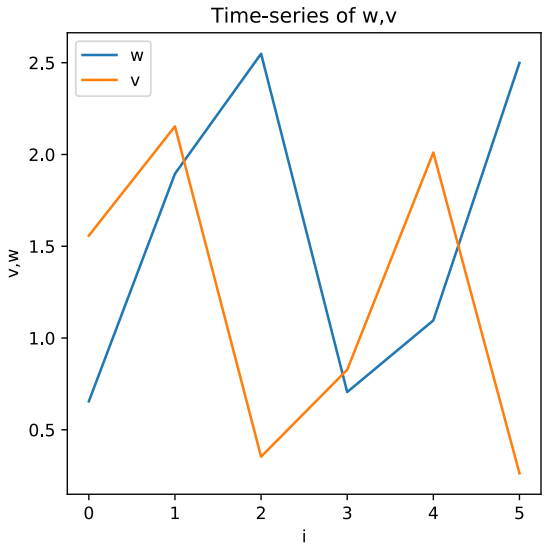


Figure 5.5: Graphical illustration of $\mathcal{D}$

v(1.86)	v(1.25)	v(0.59)	v(1.42)	v(1.14)	w(1.27)	w(2.22)	w(1.63)	w(0.9)	w(1.8)	y	v(1.86)_1	v(1.25)_1	v(0.59)_1	v(1.42)_1	v(1.14)_1	w(1.27)_1	w(2.22)_1	w(1.63)_1	w(0.9)_1	w(1.8)_1	y_1
0	0	0	0	0	0	1	0	0	0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0	0.0
1	1	1	1	1	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
1	1	0	1	1	1	1	1	1	1	0.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0
0	0	0	0	0	1	1	1	0	1	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0
1	1	1	1	1	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	1.0	1.0

Table 5.8: Example dataset with continuous variables v and w, discrete target y and a first order Markov assumption.

- $y(i) = \begin{cases} 1 & \text{if } w(i) > 1 \\ 0 & \text{otherwise} \end{cases}$

with $T = 3$ the periodicity, and $\mathcal{N}(\mu, \sigma)$ being noise sampled from a normal distribution with mean μ and standard deviation σ. We take six samples, so $i \in [0, 5]$. An illustration of an instance of this simulation is shown in Table 5.7 and Figure 5.5.

We first create all the boundaries between the variables as before and then shift them to encode a DBN. Since we have six rows, two continuous variables, and one discrete target, we will create $(6-1) \cdot 2 + 1 = 11$ columns by creating the boundaries and add another 11 for shifting to produce a total of 22 columns. This is shown in Table 5.8. As the computational complexity has increased a lot, we will restrict the structure learner to using at most six parents for each node, as the running time was becoming unfeasibly high for higher amounts of parents. This could return a sub-optimal result, but otherwise, even this simple example would be infeasible.

The result of the structure learning process is depicted in Figure 5.6. It shows that our target y takes three nodes as parents: $w^{2.22}_{t=-1}$, $w^{0.91}_{t=0}$, and $v^{0.59}_{t=-1}$. These points cut the data on three boundaries into the four distinct parts. Note that since y has been generated from w and w from v, the most logical graph would have edges from v to w and from w to y. Since the learner only has six data points to learn from, it is to be expected that the learner makes some mistakes in assessing relationships.

We can now use the three boundaries from the network that we have learned as a discretization strategy for $\mathcal{D}$ to create $\mathcal{D}'$, as shown in Table 5.9. From this dataset, as we have seen in Chapter 3, we can learn a C-DBN (shown in Figure 5.7).

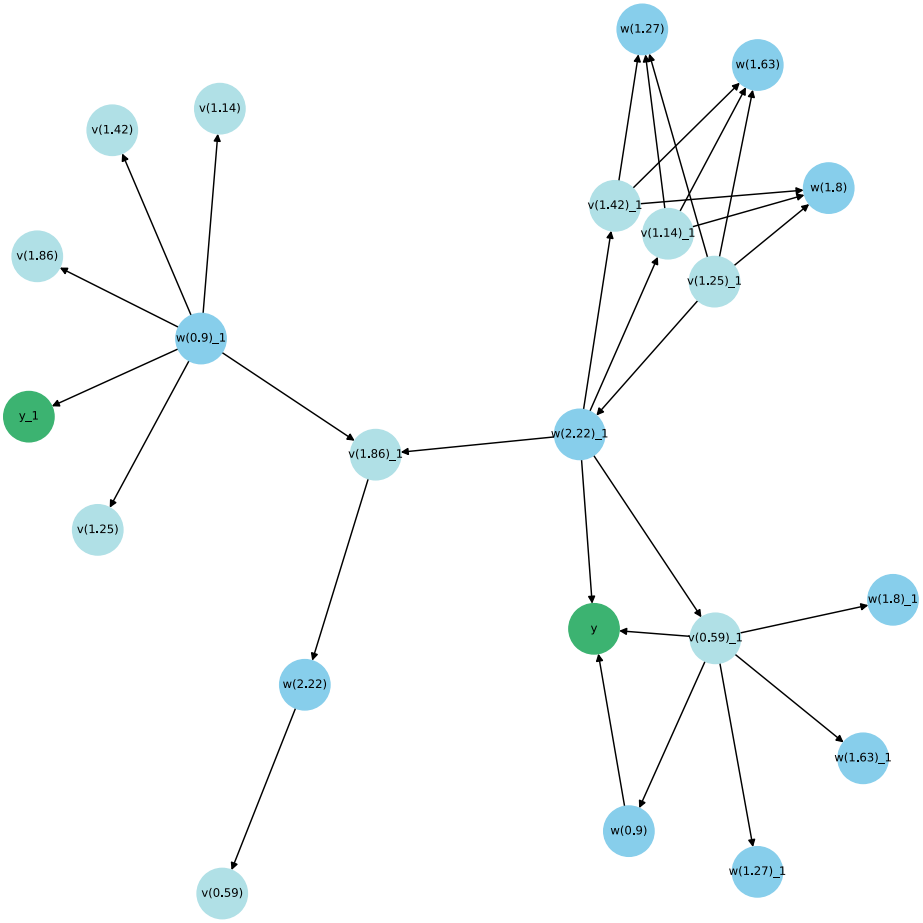


Figure 5.6: Visual representation of the learned discretization network with continuous variables v and w, discrete target y and a first order Markov assumption.

v	w	y	v_1	w_1	y_1
1	1	1	1	0	0.0
0	2	1	1	1	1.0
1	0	0	0	2	1.0
1	1	1	1	0	0.0
0	2	1	1	1	1.0

Table 5.9: Representation of the discretized dataset $\mathcal{D}'$.

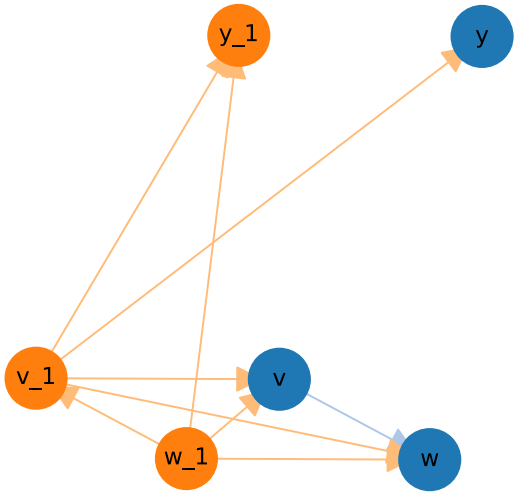


Figure 5.7: Learned C-DBN based on discretized dataset $\mathcal{D}'$.

5.6 n^{th}-order Dynamic Bayesian networks with multiple continuous variables

Since we did not use any of the variables connected to the target node in time frame $t = -1$ directly in the previous section, we can easily extend the approach from the previous section to any n^{th}-order C-DBN. We only need to unroll the network further and allow for a larger lookback, but the constraints as mentioned in Section 3.2 remain the same.

A significant impact of generalizing the order of the Markov assumption is the increased complexity. Where before we needed $2^{2\cdot(m-1)(n-1)}$ data points for learning, this now generalizes to $2^{t\cdot(m-1)(n-1)}$, where t is the number of time frames we use in our DBN. As we already had to severely limit our amount of data points for the previous examples to make it run within a reasonable amount of time, we will not even attempt to create an example for the case where $t > 2$. While interesting in theory, the complexity of this algorithm has become too high to be feasible in practice.

5.7 Example

To show the power of full automatic discretization, we end this chapter with an example using Full Automatic Discretization for a classification task with a Bayesian network. Since the computational complexity of the proposed method is too high for most real-world problems, we opted to use a single dataset that shows the reader the potential of our approach rather than doing extensive performance testing.

The Exercise Dataset

To this end, we have chosen the `exercise` sample dataset from the Python-based Seaborn library [1]. This dataset consists of 90 records, and four variables: *diet*, *pulse*, *time*, and *kind*. The first rows of the dataset are shown as an illustration in Table 5.10. *Time* refers to exercise time and is measured at either 1, 15, or 30 minutes and is therefore left as a discrete group. The *diet* variable can either be "low fat", "high fat", or "no fat", and *kind* specifies an activity, being either "rest", "walking", or "running". The *pulse* variable is the only continuous variable and ranges between 80 and 150. Its distribution is illustrated in Figure 5.8.

Method

We will try to predict the kind of activity given all other variables using two discretization approaches: Full Automatic Discretization (FAD) as described above, and the traditional unsupervised Equal Width Discretization (EWD) [14]. For FAD we have chosen to learn the structure using the BDeu score [22] with an equivalent sample size (i.e., "the strength of our prior belief in the uniformity of the conditional distributions of the network" [41]) of 20, and *at most* six parents. For EWD we have set the number of bins at seven, as that would be the maximum FAD would be able to create.

To see which method is better, we will use leave-one-out cross-validation to obtain the most information on how the methods perform on this relatively small dataset. We will use two-sided t-tests for comparing both the in-sample and out-of-sample error.

diet	pulse	time	kind
low fat	85	1 min	rest
low fat	85	15 min	rest
low fat	88	30 min	rest
low fat	90	1 min	rest
low fat	92	15 min	rest
low fat	93	30 min	rest
low fat	97	1 min	rest
low fat	97	15 min	rest
low fat	94	30 min	rest

Table 5.10: First rows of the exercise dataset

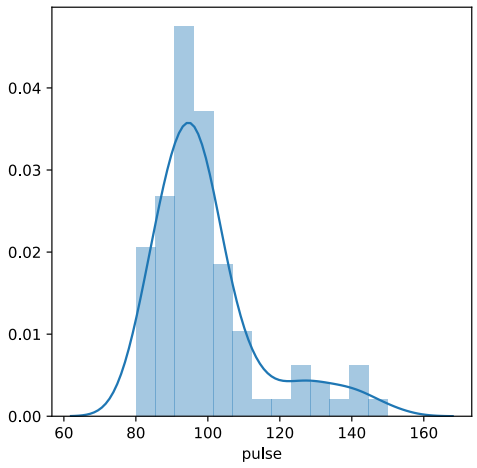


Figure 5.8: Illustration of the distribution of the pulse variable

[1] See: `https://github.com/mwaskom/seaborn-data/blob/ff48445cbd5915a4fb45bf2432015f90078c96f6/exercise.csv`

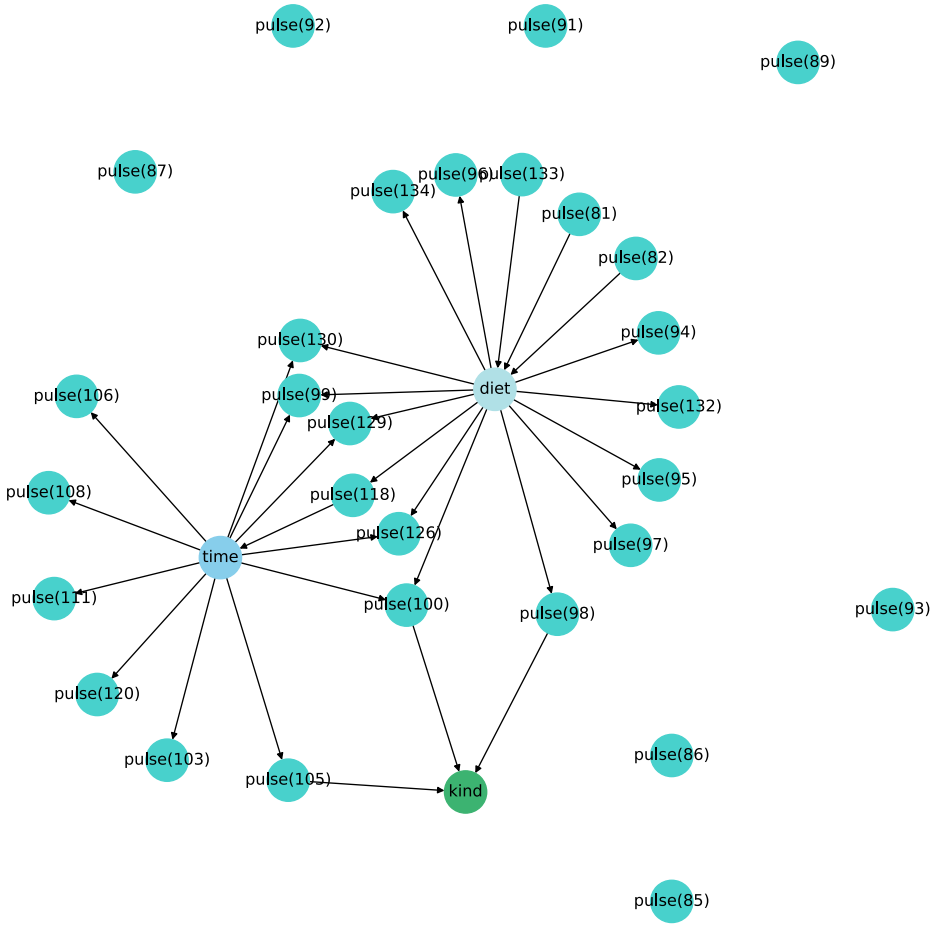


Figure 5.9: Visual representation of the learned discretization network with continuous variable *pulse*, discrete target *kind*.

Learning

For learning using FAD discretization, we first take all the unique pulse values and turn them into boundaries as described in Section 5.3. We then add the constraint that boundaries cannot be linked if they are created from the same continuous variable, and we say that they cannot receive our target (*kind*) as a parent. Next, we used the GOBNILP program to learn the structure of the network, given our augmented dataset and the set of constraints. The result is shown in Figure 5.9. It shows that our target, *kind*, uses four boundaries, thereby splitting the *pulse* set in five bins.

Now we have the FAD discretization strategy, we can make two new discretization datasets: one made by FAD and one with EWD. These datasets can subsequently be fed to the structure learner again to create two Bayesian networks, which are shown in Figures 5.10 and 5.11. The only difference between the learned structures is that the one based on FAD has an extra edge from *time* to *pulse*. This seems to correspond to a weaker relationship (as shown in Figure 5.12) than the one between *pulse* and *kind*, but using FAD we were able to pick up on it.

Having the graphs, we only have one more step to go: learning the parameters for the variables using the discretized datasets. Note that even if the structure learner would have returned the same result in both cases, having a different discretization strategy could still affect the

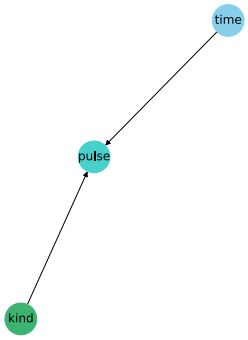


Figure 5.10: BN learned from FAD

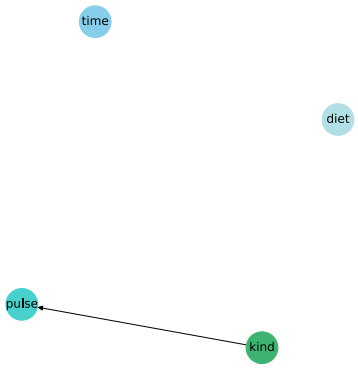


Figure 5.11: BN Learned from EWD

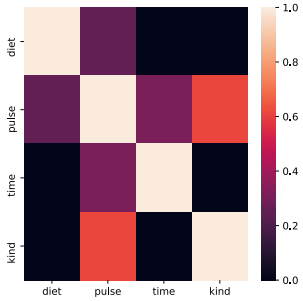


Figure 5.12: Pearson correlations in original dataset

parameter learning part of the process. For this, we used the PGMPY library [4] in Python. With this information, we can use a MAP-query to find the most likely value for *kind* given the value of all other variables.

Results

After running both algorithms on the dataset, we have obtained an in-sample accuracy of $\mu = 0.601$ with $\sigma = 0.016$ for FAD, and one of $\mu = 0.580$ with $\sigma = 0.0268$ for the EWD-based BN. This means that we have a significantly better performing model coming out of the FAD-method ($n = 89$, $p < 0.001$). Based on the out-of-sample accuracy however, there is no significant difference: while FAD achieved a lower accuracy ($\mu = 0.522$) than EWD (0.576), the standard deviation was too high ($\sigma = 0.502$ and $\sigma = 0.495$ resp., $p = 0.371$).

Analysis

These results show us two things: while there is certainly potential in FAD (as indicated by the better in-sample accuracy), it suffers when presented with new data. While it was not significantly worse than EWD, it was also no longer the better approach. First of all, it would make sense that FAD is at least as good in-sample as EWD as it has the power to construct the same model as EWD in the worst case, and improve on it if possible. Our result validates our ideas there.

There is also a likely explanation for why FAD is no longer better out-of-sample: as the approach is more powerful, it is also more prone to overfitting. This is a well-known phenomenon referred to as the Bias-Variance trade-off. To mitigate this problem, adding data is often helpful. In our circumstance, however, that will not necessarily be the case, as more data could push our approach towards computational infeasibility. To work around that problem while also making use of the potential in FAD, we will look at a heuristic approach in the next chapter.

Chapter 6

Reduced Automatic Discretization for C-DBNs

In the previous chapter, we looked at a novel approach for automatic discretization that pushes the problem of finding the right cutting boundaries towards the structure learner. We accomplished this by creating a node for all cutting options and discretizing based on the resulting selection set. While this approach seems promising, its exploding computational complexity and need for data make it impractical for most real-world applications.

In this chapter, we will use the ideas from before to construct a new approach which also reduces end-user complexity by embedding the discretization problem into the structure learning phase, while reducing the number of boundaries to check to maintain computational feasibility. While this probably means that we will not be able to achieve the same level of performance, the trade-off might be worth it since the reduced complexity does make this method applicable to real-world situations. We will first introduce the idea of Selection-based Reduced Automatic Discretization (S-RAD), then compare it on theoretical bases to a closely related approach, describe the required transformation for learning and finish with an example.

6.1 Selection-based Reduced Automatic Discretization (S-RAD)

To reduce the number of boundaries that we need to check, which causes FAD to be infeasible, we can use a heuristic that provides us with a range of possibly interesting boundaries. We would then limit our search for discretization strategies to the boundaries produced by the heuristic. While we could construct such a heuristic on our own, we thought it wise to use existing discretization methods. Therefore, our approach will be to generate a large but fixed number of cutting points using methods such as Equal Width Discretization, Equal Frequency Discretization, and others to create potentially good cutting points. Our proposed discretization technique using the structure learner will then use these options to select the best possible discretization strategy.

There are multiple ways of creating a discretization strategy from heuristics using a structure learner. One intuitive variant involves generalizing the concept of discretization nodes as introduced in the previous chapter. Since many discretization algorithms have hyper-parameters, we can create a number of discretization nodes that describe the different *instantiations* of these methods and use automatic discretization to *select* the best version. We will, therefore, call this method *Selection*-based Reduced Automatic Discretization (or S-RAD).

6.2 Selection vs. Estimation

S-RAD might at first appear to be very similar to an existing method of optimizing discretization methods: the common machine learning practice of hyper-parameter estimation [11]. Hyper-parameter estimation involves testing the various options for algorithms (either complete models or parameters) using an additional dataset, and selects the option with the highest accuracy on this dataset. There is a key difference: while hyper-parameter estimation maximizes the accuracy of a classification task, S-RAD's approach takes a step back and tries to balance this with finding the best underlying model.

To find the best underlying model, we need to have a way of measuring the quality of such a model. This is a problem we faced before when we tried to find the best structure for a (C-)DBN given a dataset. We then solved this by using a structure score which represents the likelihood of a model given the dataset. Here we can follow a similar approach: we will try to find the discretization strategy which helps maximize the structure score, thereby creating a model with the highest possible likelihood.

Note that this is the opposite approach of hyper-parameter estimation. In hyper-parameter estimation, we expect that if we maximize accuracy on a validation set, we will have created the best possible underlying model. Here we want to create the best underlying model and expect that this will also produce a good accuracy. In theory, if we had no noise and infinite amounts of data, both approaches would work well. However, since we do have noise and our supply of data is not infinite, we have to choose what is more important: accuracy or fit. Since the reason we have chosen Dynamic Bayesian networks in the first place is to make temporal processes explainable and insightful, we will prefer to maximize our fit. The next section will describe how we will achieve this by building on the techniques from FAD.

6.3 Transformation

To embed the problem of discretization into that of structure learning, we need to extend our definition of a discretization node. Where before we said that a discretization node d_v^r for variable v was binary, so it partitions v into two parts based on boundary r, we now let d_v take on multiple values. Each of these values indicates a bin of an existing discretization approach. Different discretization nodes then indicate the various combinations of algorithms with their instantiations. We use a similar definition for selection as in Definition 5.2: if a discretization node is the parent of the hidden node (i.e., the target), it is selected. We do need an additional constraint to guarantee that we have a single strategy: the selection set must have a size of at most one:

Constraint 6.1 (Singular Discretization Strategy[1])**:**
The selection set S_v for node v must contain one item: $|S_v| = 1$

The approach to S-RAD is shown as pseudocode in Listing 6.1. For the input, we assume to have a dataframe or dataset consisting of the different variables on the columns and values in the rows. In addition, we have a list of discretization variables, a list of the different algorithms and instantiations referred to as *discretization methods*, the target variable of the network, and the regular DBN parameters.

[1]In our experiments, we have used the constraint that $|S_v| \leq 1$. While this means we can sometimes find no discretization strategy, this did not often occur in practice. In the cases where it did happen, we discretized the dataset to a constant value. This is also what is shown in the pseudocode.

```
def S-RAD(dataframe, discr_variables, discr_methods, objective, look_back, parent_lim, ESS):
    for var in discr_variables:
        for method in discr_methods:
            dataframe[var_method] = method(dataframe[var])
        dataframe.drop(var) # remove the continuous variable

    dbn_dataset = create_lookback_dataset(dataframe, look_back)
    G = structure_learn(dbn_dataset, parent_lim, ESS)

    strategy = {}
    for var in discr_variables:
        for method in discr_methods:
            for timeframe in range(look_back):
                if (var_method_timeframe, objective) in G.edges:
                    # select method of most recent timeframe
                    if not strategy[var_method]:
                        strategy[var_method] = method

    return strategy
```

Listing 6.1: Pseudocode for using Selection-based Reduced Automatic Discretization

The algorithm will first create a new column for every variable and every discretization method with the result of discretizing the variable according to this method. After that, we remove the original variable so the learner can not use it for additional information. Next, we can transform the dataset into a DBN problem as before, and learn its structure.

The discretization strategy that we choose is the one belonging to the variable with the highest time frame (being closest to the present). Note that this choice is based on simplicity but can be changed to a more sophisticated variant based on e.g., a weighted average if the events in the far-past tend to have more influence than the near-present, or if a large number of timeframes is used.

6.4 Example

To illustrate how S-RAD works, we will now look at an example using a subset of the Ottawa Bike Counters dataset [2]. This dataset records the number of bikers on a given road for every day, measured per day for eight years. Aside from the day of the week, it also includes the mean temperature and the amount of snow on the road. We discretize the number of bikers by hand using Equal Frequency Discretization (to avoid skewing our model towards one group) into three parts [2].

For this example, and in fact, for the experimentation later on, we will limit ourselves to discretizing only a single variable, even though our approach would extend to a combination of nodes. Since we have two continuous variables in the dataset, we will discretize one of these variables, the amount of snow on the ground, ourselves (using EFD with 10 bins). A part of the resulting dataset is shown in Table 6.1.

We will now apply S-RAD to the mean temperature in this dataset so that we can predict the number of bikers. We want to select from five different discretization methods: discretizing based on the median value, EWD with either two or three bins, and EFD with either two

[2]Note that we do not use automatic discretization for this variable because it is our classification target. It should only depend on the purpose of the model.

Temp	Snow	Month	Bikers
-13.1	2	2	0
-6.9	2	2	0
-3.4	2	2	1
2.9	1	2	1
-4.8	1	2	1
-7.4	1	2	1
-4.1	1	2	1

Table 6.1: Sample taken from the Ottawa Bike Counters dataset

Temp	Snow	Month	Bikers
0	2	2	0
0	2	2	0
0	2	2	1
0	1	2	1
0	1	2	1
0	1	2	1
0	1	2	1

Table 6.2: Ottawa Bike Counters dataset discretized using the selected EFD-3

Snow	Month	Bikers	T-Median-0	T-EFD-2	T-EFD-3
2	2	0	0	0	0
2	2	0	0	0	0
2	2	1	0	0	0
1	2	1	0	0	1
1	2	1	0	0	0
1	2	1	0	0	0
1	2	1	0	0	0

Table 6.3: Discretization dataframe produced from Table 6.1 by discretizing *Temperature* according to five different methods.

or three bins. We do this by creating additional columns as in Listing 6.1 for each possible method by applying each method on the *temperature* variable. Finally, we remove the original continuous variable to get Table 6.3.

Now we have a dataset containing all discretization variables, we can learn a DBN as before. In addition to the three standard constraints which specify that nodes cannot have parents in the future, that the hidden variable cannot be connected to its past, and that the objective cannot have any children, we also add the constraint that only one of the discretization variables can be the parent of the objective. If we would not enforce this, the structure learner could suggest a combination of discretization tactics that would increase complexity. This last constraint is, however, not entirely trivial to implement for the Gobnilp solver. The implementation of the constraint is described in more detail in Appendix B.

After having implemented all the constraints, we can run our structure learning procedure. A graphical overview of the resulting structure is shown in Figure 6.1. From the structure, we can see that the discretization strategy that relies on EFD with two bins has been selected, and is, therefore, our choice for discretization. Note that the structure learner has found that its representation is best when it uses the temperature from the previous timeframe, rather than from $t = 0$. We will not create any constraints to prevent this from happening since this selection indicates that most information can be obtained from the previous timeframe, which the final DBN can replicate.

Since we have found a discretization strategy, we can go back to the original dataset and discretize the temperature variable using this method alone. A sample from the dataset is shown in Table 6.2. From this dataset, we can learn our final DBN for predicting the number of bikes as we have shown in Chapter 3. The final graph is shown in Figure 6.2.

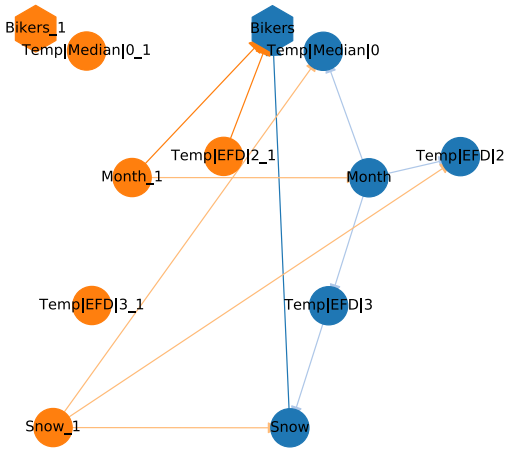


Figure 6.1: Visual representation of the learned discretization DBN for the bike counters dataset. The number of bikers, show as hexagons, are the target nodes.

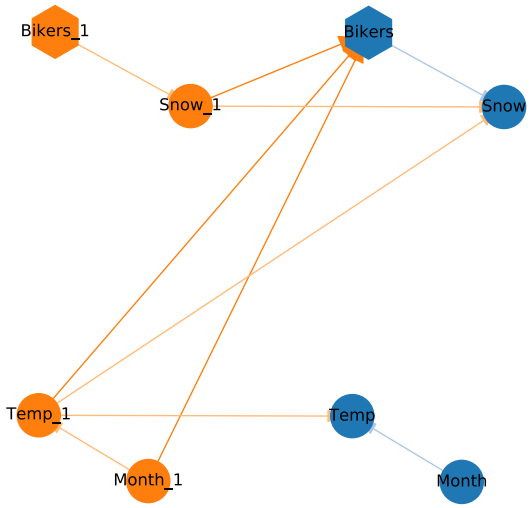


Figure 6.2: Learned DBN based on discretized bikecounters dataset

Chapter 7

Experimental Verification

In this chapter, we will test our proposed method of classifying time-series by C-DBNs with S-RAD. To that extent, we use two real-world datasets: *Ottawa Bike Counters* [2] and *Room Occupancy* [1], as well as a synthetic one. The real-world datasets were selected from a broader range of potential options. See Appendix A for an outline of the reason why some datasets were not included. Here we will first discuss what we want to discover, then we provide an overview of the datasets that are used, and conclude with a description of the set up for the experimentation process.

7.1 Hypothesis

Throughout this paper, our goal has been threefold: we want to learn a model *automatically*, which is *accurate*, and *explainable*. Now we have constructed our proposed approach, we want to test if our method fulfills these requirements. To that extent, we will compare our method based on two scores in two types of situations.

First, we want to verify that automatically learning a discretization strategy using S-RAD is consistently either better than or on par with using discretization without prior knowledge. As the alternative discretizer, we have selected the median discretizer. This comparison with the median is informative to us because the median is often used when nothing about the data is known in advance. To test consistency, we will look at discretizing different variables in the same dataset. Second, we want to compare S-RAD against the discretization strategies that it uses as heuristic: EWD, EFD, IQR, and Median. Since S-RAD can choose directly from these heuristics, we think that none of these methods will consistently outperform S-RAD.

To measure the performance of our model, we will use three scoring functions. To start, we note that accuracy encompasses two different concepts in our C-DBN system. First, we want our prediction of the *target value* to be correct. This is measured using the accuracy score: the number of correct predictions divided by the total number of predictions:

$$\text{Accuracy}(Y, \bar{Y}) = \frac{1}{N} \sum_{i=0}^{N} \begin{cases} 1 & \text{if } Y_i = \bar{Y}_i \\ 0 & \text{otherwise} \end{cases}$$

in which $\bar{Y}$ is the set of predicted values, Y is the set of true values, and N is the size of both sets.

Then we also want the *probability* which we assign to each outcome to be accurate. We will use the multi-variate Brier score [6] to represent this wish to measure predictive accuracy [20]. The Brier score indicates how close our probability estimates are: the higher our probability is for the true classification value and the lower the probabilities we assign to incorrect values, the lower the Brier score. A *low* Brier score therefore indicates a *good* result:

$$\text{Brier}(P, E) = \frac{1}{N} \sum_{j=1}^{R} \sum_{i=1}^{N} (P_{i,j} - E_{i,j})^2$$

where N is the number of classified feature vectors, R is the number of assignments the target variable can take on, $P_{i,j}$ is the probability the model computer for seeing assignment i for features j, and $E_{i,j} = 1$ if the true assignment for j is i or 0 otherwise [6].

Now we can state our hypotheses as follows:

1. S-RAD consistently performs either the same or better than using the **Median** discretizer, based on the accuracy and the Brier-score.
2. There is no discretizer in our test set that can consistently outperform S-RAD based on accuracy and the Brier-score

7.2 Datasets

To test our hypotheses, we will use three different data sets. In this section, we provide some background information and descriptive statistics to get familiar with their properties.

7.2.1 Ottawa Bike Counters

The original Ottawa Bike Counters dataset, created for [8] and made available by [15], contains 13 columns and 28,786 rows. Its purpose is to measure the number of bikers on a certain road. To help with prediction, it also includes weather conditions. Most columns (7) are derivatives of the temperature, amount of snow, or amount of rain. Other variables include location, amount of bikers, and date. For our tests, we only look at one location (id = 2). Since we want to predict a discrete variable, we create three groups of the number of bikers using EFD (which ensures that our model is balanced). We also scale down the number of columns to reduce complexity. To measure time, we convert *day*, *day of year*, and *day of week* to a single variable: *month*. For summarizing the weather conditions, we will only use *mean temperature* and *snow* since these two have a strong relationship with the number of bikers ($\rho = 0.73$ and $\rho = -0.5$ resp.). The correlations between all variables are shown in Figure 7.2.

For this dataset we create two types of experiments: one in which we only try to predict the hidden variable from one of the two probabilistic variables (either *snow* or *mean temperature*), and one in which the network combines the two. For the variable that is not automatically discretized, we used EFD with five bins. We chose EFD for this purpose because it produces bins of equal size, which avoids creating unbalanced classes that might clash with our frequency learner. Five bins do reduce some complexity, but also allow quite some flexibility for our model to preserve information. The original distributions are shown in Figure 7.1.

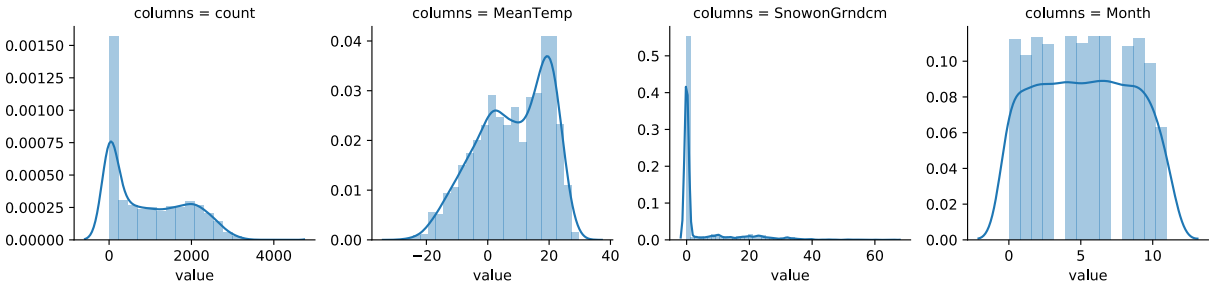


Figure 7.1: Histogram and Kernel Density Estimation (KDE) of the target and variables to be discretized for the Ottawa Bike Counters dataset

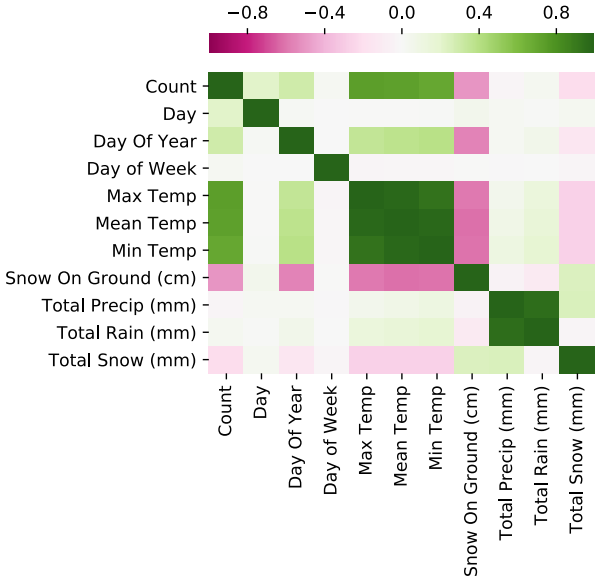


Figure 7.2: Heatmap showing the Pearson correlations between the variables in the original Ottawa Bike Counters

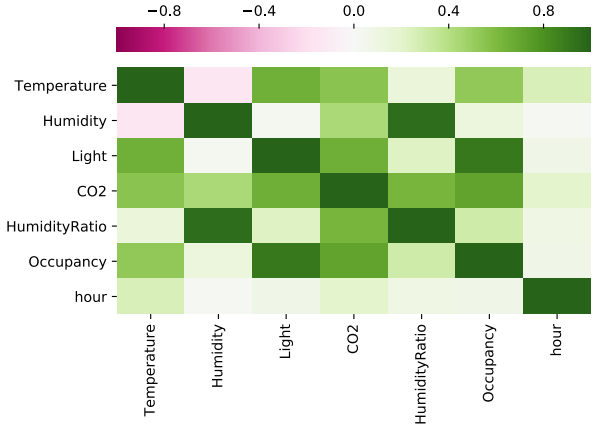


Figure 7.3: Heatmap showing the Pearson correlations between the variables in the original room occupancy dataset

7.2.2 Room Occupancy

The room occupancy dataset contains 7 columns and 8143 rows. The variables include the *date*, *temperature*, *humidity*, *light*, *CO2*, *humidity ratio* and *occupancy* of a room. The goal is to predict whether someone was present in a room given the six other observations. All measurements were conducted once a minute. For testing, we turn the date into the current hour to avoid having individually identifiable datapoints. To reduce complexity, we remove the *humidity ratio* variable (which contains the same information as the *humidity* variable), as well as the *temperature*. That leaves us with three variables for predicting room occupancy.

Of the three remaining variables, *light* has the strongest correlation with occupancy: $\rho = 0.91$. Second is *CO2*: $\rho = 0.71$. The weakest (linear) relationship is between *humidity* and *room occupancy*: the Pearson coefficient is in that case only 0.13. The correlations between all variables are shown in Figure 7.3. The distribution of the variables that we are either going to predict or discretize are estimated and graphically shown in Figure 7.4.

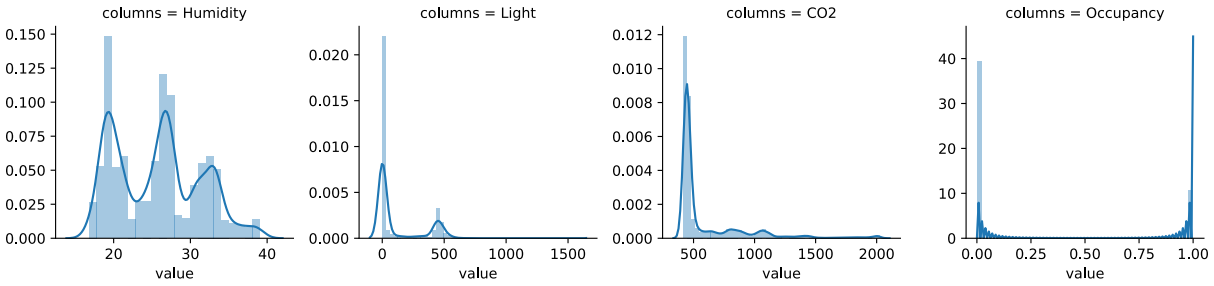


Figure 7.4: Histogram and KDE of the target and variables to be discretized for the room occupancy dataset

7.2.3 Synthetic Sine-based

The last dataset we use is a synthetic one, based on the example in Section 5.5 with some slight modifications. As before, we have two features: v and w. We define v as:

$$v(i) = \sin(iT) + \epsilon + 1$$

with T the periodicity and ϵ noise generated from a normal distribution with $\mu = 0.2$ and $\sigma = 0.2$. We define w based on v:

$$w(i) = v(i-1) + \sigma$$

where σ is noise generated from a normal distribution with $\mu = 0.4$ and $\sigma = 0.1$.

Finally, we create the target node y from a static threshold:

$$y(i) = \begin{cases} 1 & \text{if } w(i) > 1.5 \\ 0 & \text{otherwise} \end{cases}$$

This means that both y and w are tracking v. If we want to simulate a dynamic human decision process for y, this is a sensible approach since that also involves reaction time. This relationship becomes visible in the correlation diagram in Figure 7.7. Both w and y have a relatively strong positive correlation as y reacts in the same time frame as w. There is a much weaker correlation between y and v since the decision for y goes through w and has to incur two different noise factors. Since the two operate with a lag of a single timeframe (and a periodicity of three), the relationship becomes slightly negative. The connection between w and v is for the same reason negative, but a bit stronger as the two are directly related.

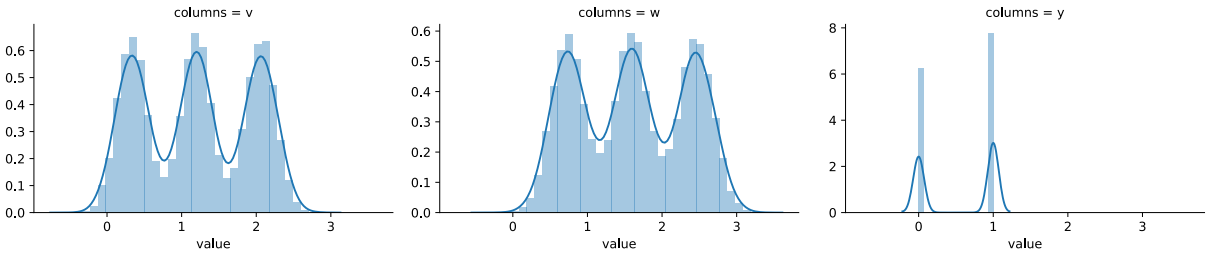


Figure 7.5: Histograms and KDE for v, w, y in a sample of 20,000 rows

The sine function itself is visible in the histograms in Figure 7.5. Since we have many data points, we can see the original distributions through the noise. Finally, the amount of data

points for $y = 0$ is slightly lower than that for $y = 1$ (they occur in a ratio of about $9 : 11$), which is caused by the mean of 0.1 for the noise factor of w.

In the test cases, we will use a periodicity of $T = 3$ to make sure we have a highly dynamic dataset with many unique data points. Also, to make sure we have enough data points, we sample $20,000$ rows. The first five periods are shown in Figure 7.6.

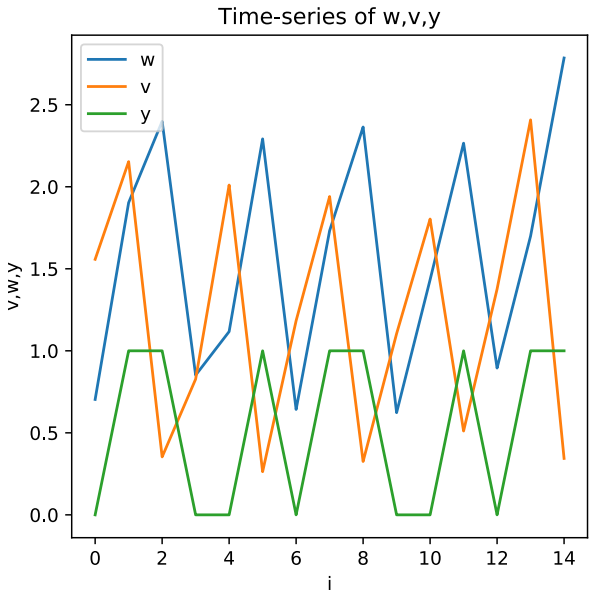


Figure 7.6: Plot of first 5 periods that have been sampled for v, w, y

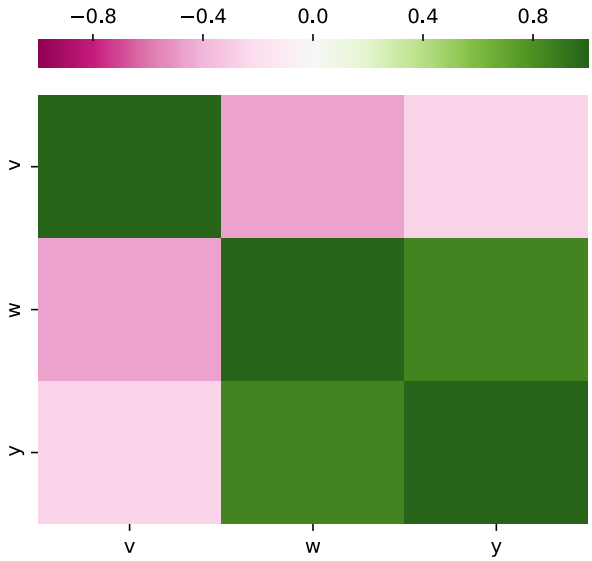


Figure 7.7: Visual overview of Pearson correlations between v, w, y in a sample of 20,000 rows

7.3 Testing Set-up

We have conducted our experiments using Gobnilp 1.6.3, PGMPY `c34ccd-dev` and Python 3.7.3 on Ubuntu 18.04.2 on both a desktop computer and an AWS EC2 server. Our full testing environment is described in Appendix D.

For each dataset, we learn a C-DBN with an effective sample size of 20 and a parent limit of 5 (or the maximum number of variables in the graph if that amount is lower). We use a first-order Markov assumption: we have one timeframe at $t = -1$ and one at $t = 0$. In each test case, we provide the network the value for all variables as evidence, except for the target variable in $t = 0$. S-RAD was composed of eight heuristics: IQR, Median, EWD-2, EWD-3, EWD-10, EFD-2, EFD-3, and EFD-10.

For all three datasets, we test the effect of discretizing one variable, while dropping all other continuous variables from the dataset. For the bike counters dataset, we also test the effect of leaving in the other continuous variable but discretizing it manually using EFD-5.

7.4 Cross-validation with temporal data

Since we want to provide the best overview of our dataset, we use k-fold cross-validation. This method runs k tests: it divides the dataset into k blocks and for each run selects one fold as the test set, and uses the other folds for training. In this way, we can test our entire dataset for

training as well as testing. Since we use $k = 10$, we will have ten test sets for each experiment. We can then report on the different scores by taking the average of these ten results.

However, as we are dealing with temporal data, we cannot use k-fold cross-validation directly. Cross-validation assumes that all measurements are independent of each other, and for us, that is not the case. If we would do so, we would possibly pollute the train set with data that is strongly related to our test set, thereby making our results inaccurate. To break these dependencies, we use ideas from a method developed by Racine for the field of Econometrics called h, v-fold cross-validation [38]. This method involves "padding" the h (which is denoted by k in our context) blocks of test data with v items that we will not use for either training or testing. If we choose a large enough value or v, we can assume the trainset and testset to be independent.

While Racine does offer some guidelines for choosing a good value for v, the author notes that this is application-specific. The guideline says that v should be about $1/4^{th}$ the size of k. This rule makes the padding depend on the size of our dataset, not on the characteristics of the data. While it might work well for model selection, as the author indented initially, it can be either overly strict or lenient for our purposes of breaking dependencies for a representative score. Therefore, instead of using the suggested v, we take an alternative approach.

We base our approach on the properties of the data itself, using the partial autocorrelation function (PACF). The PACF specifies after how many time frames (or *lags*) a variable is no longer correlated with itself anymore [5, p. 64] [18]. This function corresponds exactly to what we want: the number of time frames after which a timeframe is no longer dependent on previous data.

There are two ways to find a cut-off point after which we deem an auto-correlation irrelevant. First, if the correlation is significantly less than 0.1, it is only a very soft correlation that will, in many instances, be surpassed by different variables in the network. Second, we compare it against the correlation between the variable and *the smallest time-based variable*. Since the time-based variables are always given for each timeframe (as they are deterministic, see Section 3.2), the effect of auto-correlation becomes irrelevant when the correlation between the time-based variable is higher than the auto-correlation. When the auto-correlation after i timeframes has become smaller than both of these values, we assume that we need at most i timeframes to break independence between any set of timeframes. Hence, we can use the value of i for v in k, v-fold cross-validation.

For calculating the PACF, we have used Python's `statsmodels` library. To split the data into padded blocks using k, v-fold cross-validation, we have created an open-source Python library, available on GitHub: `https://github.com/daanknoope/kv_block_cross_validation`.

7.5 Statistic tests

After we run our experiments, we will use the paired t-test to compare discretization methods, since we are using the same datasets for all the algorithms. We say that a difference is significant if $p < \alpha$, and $\alpha = 0.05$ as usual.

Since it is our goal in here to provide an understanding of when S-RAD can be useful, we will use many t-tests to compare performance. As it is not our goal to provide the definitive proof for its superior performance in all situations, we do not compensate for the extensive amount of significance testing using a correction that would otherwise be required, as that would be overly harsh and provide us with little insight.

Chapter 8

Experimental Results

In this chapter, we show the results of the experiments we discussed in the previous chapter. We split this into four parts: the results based on the room occupancy dataset, the results for the bike counters dataset in which there were either one or multiple variables to learn from, and finally those from a synthetic dataset. Additional results are shown in Appendix E.

8.1 Room Occupancy

In the room occupancy dataset, there were three variables to be discretized: *humidity*, *CO2*, and *light*. The barplot in Figure 8.1 shows the accuracy and Brier-score, based on S-RAD and the eight other discretization methods it is composed of. All variables were tested in isolation: if *humidity* was tested for example, *CO2* and *light* were removed from the dataset.

It is important to note that for the *light* variable, no results are shown for the various versions of EWD. This lack of results is caused by the fact that EWD can, and in the case of *light* will, create empty bins. The structure learner has trouble with using empty bins since it means that the arity of variables need not be the same as the number of unique variables. This incompatibility could be fixed in later versions, but here EWD has been excluded, and we used S-RAD without EWD as a heuristic.

From the bar plots, it becomes clear that the overall score on the training set is much higher than that on the test set. Also, results based on the train set have a much lower variance than those based on the test set. While it is not uncommon for models to perform better in-sample than out-of-sample, the difference here is rather striking.

We suspect that there are three potential causes at play here. First: the train sets are much larger than the test set, and often largely overlap between tests (due to cross-validation), driving down the variance between experiments. Second, since the data is dynamic, it could be that the training set does not represent the test set very well. While we sample from the same dataset, since time series are not Independent and Identically Distributed (IID), we might create a train set that is generated by a different underlying distribution than the test set. Using synthetic tests in which we control for the underlying distribution can help here, as we will see later on. Third, the number of measurements in a time-series can be misleading. While the overall dataset consists of around 81,000 rows, there may be many duplicate measurements in which only the time changes, but not the other variables. In other words, part of the time series can be very *stable* and contain a rather low number of change points. If our model by chance happens

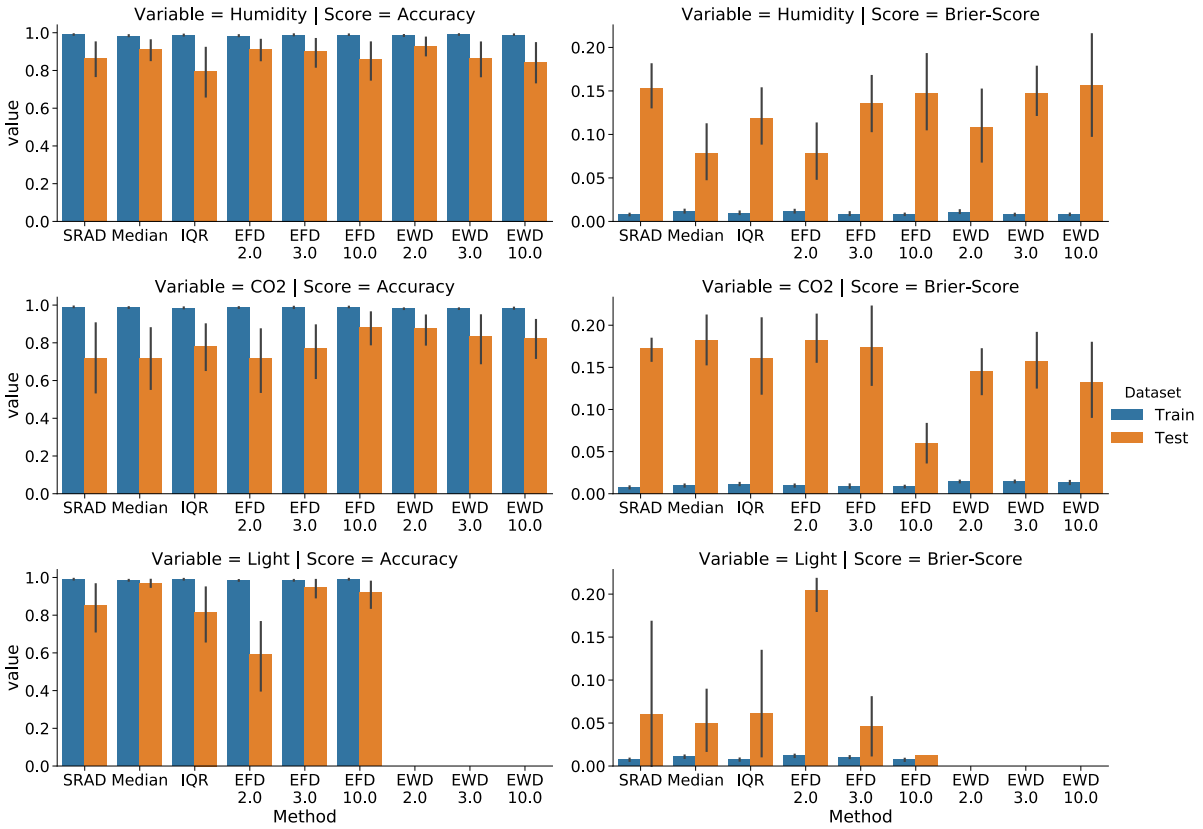


Figure 8.1: Barplot with 95% confidence intervals of the results of the three discretized variables together with the accuracy and the Brier-score for the room occupancy dataset.

to perform poorly in one situation that remains stable for many measurements, score functions like the accuracy and Brier-score could give us a skewed impression of performance. This low number of unique datapoints would also explain the rather large variance between results in the test sets.

The full results are shown in Table E.5. From this table, we can see that in-sample, S-RAD achieves a significantly better accuracy and Brier-score than the Median discretizer for all the different variables. In the test set, the exact opposite happens: the median outperforms S-RAD in all cases, though never significantly so (due to a high standard deviation, as discussed before).

Dataset	Score	Variable	Alternative	Score SRAD	Std. SRAD	Score Alternative	Std. Alternative	p
Test	Accuracy	CO2	EFD 10.0	0.716	0.31	0.884	0.14	0.027

Table 8.1: Table showing the score types, variables and alternative strategies when the alternative was significantly better than S-RAD for the room occupancy dataset

There is no discretization strategy that consistently and significantly outperformed S-RAD in the test sets for all the variables. Only EFD-10 is significantly better than S-RAD based on the accuracy for the *CO2* score, but for the other variables there is no significant difference. These results are shown in Table 8.1.

8.2 Bike Counters

The next results we look at come from the bike counters dataset. For the previous dataset, we tested the effect of discretization based on a network with a single probabilistic variable: the one we discretized automatically. For this dataset, we will at first take the same approach and only use a single probabilistic variable (in addition to the other types of variables). After that, we use both probabilistic variables, one of which is discretized beforehand (both combinations are tested).

8.2.1 Single probabilistic variable

The first results we look are based on experimenting with C-DBNs that only contain one probabilistic variable (altering between *snow* and *temperature*). The bar plot in Figure 8.2 visually shows an overview of the results. For this dataset, the difference between the test set and the train set seems smaller than in the previous dataset. The variance remains relatively high for the test sets, however.

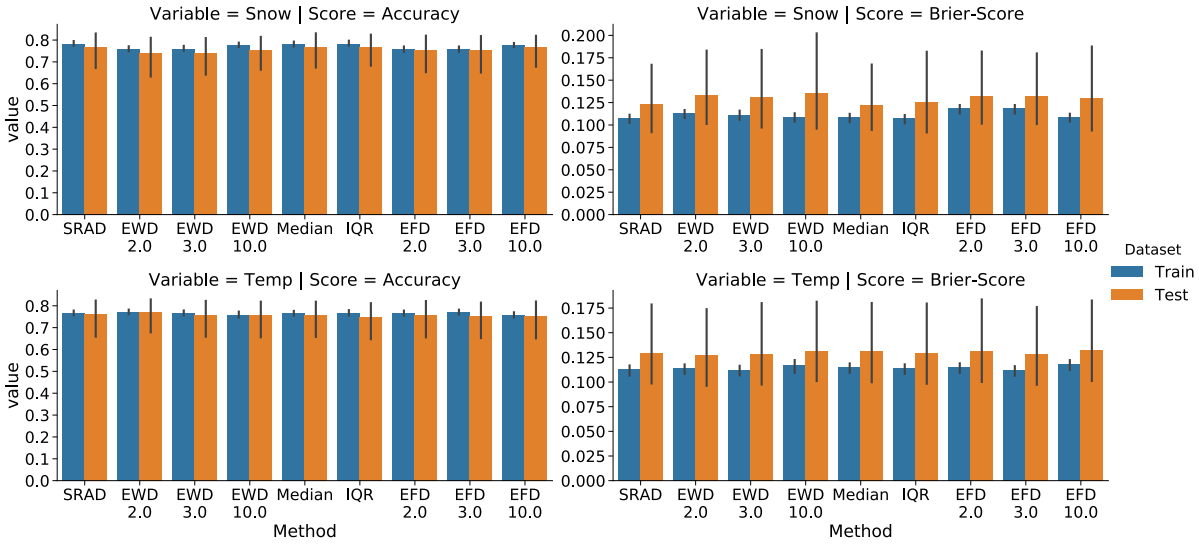


Figure 8.2: Barplot with 95% confidence intervals of the results of the three discretized variables together with the accuracy and the Brier-score for the bike counters dataset with a single variable.

Most striking is that the accuracy scores of the different discretization methods in both the test set and the train are very similar to another. This idea is reinforced by the full results shown in Table E.2. There is only a significant difference in accuracy between S-RAD and the Median discretizer for the *snow* variable in favor of S-RAD, but even so, the difference is minor (with 0.002 percentage points). The same applies to the Brier-score: all differences are either statistically insignificant or rather minor in size. This is shown in Table 8.2.

When we check for steady outperformers, we see that none of the methods consistently outperforms S-RAD in the testing data. EWD-2 and EFD-3 are both significantly better based on accuracy for the *temperature* variable in the training data, but not for the *snow* variable. S-RAD is thus not consistently outperformed in this dataset.

Dataset	Score	Variable	Alternative	Score SRAD	Std. SRAD	Score Alternative	Std. Alternative	p
Train	Accuracy	Temp	EWD 2.0	0.765	0.020	0.770	0.019	0.003
Train	Accuracy	Temp	EFD 3.0	0.765	0.020	0.769	0.019	0.005
Train	Brier-Score	Temp	EFD 3.0	0.113	0.009	0.112	0.008	0.029

Table 8.2: Table showing the score types, variables and alternative strategies when the alternative was significantly better than S-RAD for the single bike counters dataset

8.2.2 Multiple probabilistic variables

The second way we look at the bike counters dataset is by using both *snow* and *temperature* in the learning process. We only discretized one of the two automatically and used EFD with five bins to discretize the other. The results of this are shown again in a bar plot, in Figure 8.3. What stands out most here is that the scores are remarkably close to those based on the networks with only one of the two variables. While such similarities can be an indication that our model did not learn anything, that is certainly not the case. Since the target consists of three equally sized groups, a majority class predictor would get about 33.3% correct on average, while our average accuracy hovers around 78%. Adding the extra node compared to the previous dataset has just not resulted in a better predictor, meaning that one of the two probabilistic variables would be informative enough.

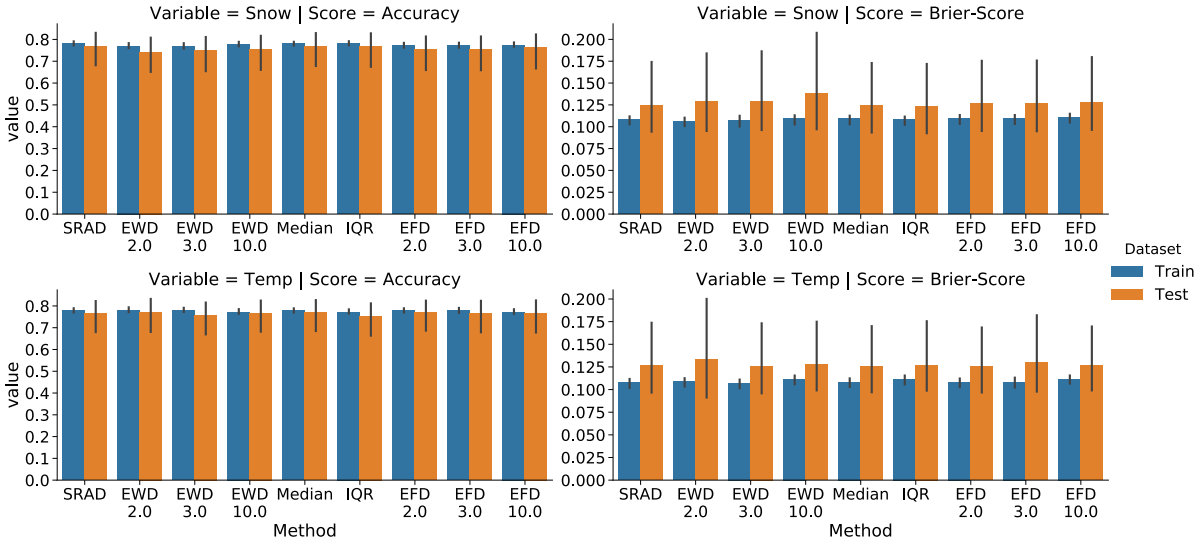


Figure 8.3: Barplot with 95% confidence intervals of the results of the three discretized variables together with the accuracy Brier-score for the bike counters dataset with multiple variables.

Our consistency results mostly overlap with those of the previous section, except for EWD-3, as shown in Table 8.3. Interestingly enough, this method was able to outperform S-RAD based on accuracy in training data. EWD-3 was only capable of significantly outperforming S-RAD based on the *Temperature* variable, and only in the training data. Hence, S-RAD is not consistently outperformed by any of the alternative discretization methods in this dataset.

Dataset	Score	Variable	Alternative	Score SRAD	Std. SRAD	Score Alternative	Std. Alternative	p
Train	Accuracy	Temp	EWD 3.0	0.778	0.018	0.780	0.018	0.005
Train	Brier-Score	Snow	EWD 2.0	0.108	0.008	0.106	0.008	0.023

Table 8.3: Table showing the score types, variables and alternative strategies when the alternative was significantly better than S-RAD for the combined bike counters dataset

8.3 Synthetic

Until now, we have seen a limited performance increase of S-RAD on C-DBNs over the other methods. In the first dataset, we were able to achieve a higher in-sample accuracy but suffered on the test sets, and in the bike counters dataset we did not see any significant difference. Although other methods did not consistently outperform S-RAD, the rather large variance between scores in the test set has made it hard to find clear differences. Therefore we have also created this last, synthetic, dataset. It contains many data points sampled from a dynamic distribution with only a limited amount of noise and many changepoints. These properties help us avoid a large out-of-sample test set. Indeed, we see in the results, as shown in the barplot in Figure 8.4 much smaller confidence intervals.

These reduced out-of-sample variances mean we can now study the effect of S-RAD more accurately. As shown in Table E.8, S-RAD ($\mu_v = 0.948$, $\sigma_v = 0.009$ and $\mu_w = 0.961$, $\sigma_w = 0.005$) is able to achieve a significantly better accuracy than when we used the Median discretizer, both for v and w ($\mu_v = 0.929$, $\sigma_v = 0.009$, $p_v < 0.001$ and $\mu_w = 0.939$, $\sigma_w = 0.007$, $p_w < 0.001$). The same is true for the Brier-score: S-RAD's average of both variables ($\mu_v = 0.035$, $\sigma_v = 0.000$ and $\mu_w = 0.024$, $\sigma_w = 0.002$) is significantly lower than that of the Median discretizer ($\mu_v = 0.058$, $\sigma_v = 0.006$, $p_v < 0.001$ and $\mu_w = 0.039$, $\sigma_w = 0.003$, $p_w < 0.001$).

In fact, S-RAD is the top performer in this dataset together with EFD-10 (for the full results, see Appendix E, Tables E.7). The two achieve the same scores, leading us to believe that S-RAD has always chosen EFD-10 from its range of heuristics for the synthetic dataset.

Dataset	Score	Variable	Alternative	Score SRAD	Std. SRAD	Score Alternative	Std. Alternative	p
Train	Accuracy	w	EWD 2.0	0.962	0.003	0.963	0.002	0.0
Train	Accuracy	w	EWD 10.0	0.962	0.003	0.963	0.002	0.0

Table 8.4: Table showing the score types, variables and alternative strategies when the alternative was significantly better than S-RAD for the synthetic dataset

EWD-2 and EWD-10 were only able to beat S-RAD on accuracy in the training set using w, but not out-of-sample and not consistently. We can, therefore, conclude that S-RAD is not a consistently worse performer based on Accuracy and Brier-score.

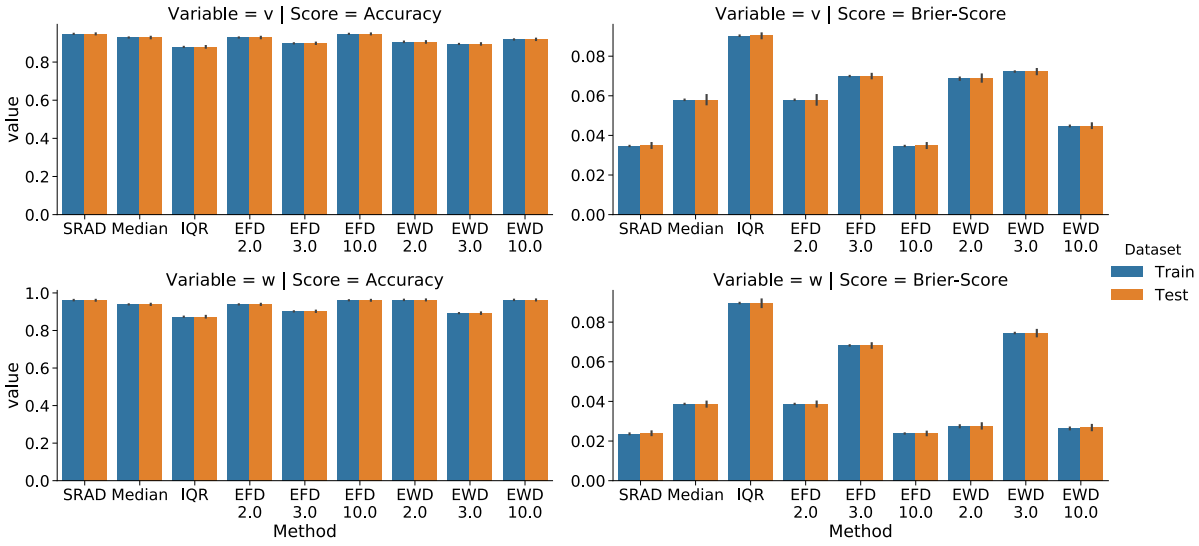


Figure 8.4: Barplot with 95% confidence intervals of the results of the three discretized variables for the synthetic dataset together with the accuracy and the Brier-score.

Chapter 9

Analysis of Experimental Results

In the previous chapter, we saw the results of running our experiments. We did these experiments to test the two hypotheses postulated in Section 7.1: 1) that S-RAD is consistently better than or on par with the median discretizer, and 2) that S-RAD is not consistently outperformed by another discretization method it is composed of. In this chapter, we analyze the results of each dataset and describe the qualities and weaknesses of S-RAD.

9.1 Room Occupancy

Our first hypothesis was that S-RAD would consistently either outperform or perform on par with the discretizer based on the median. To measure this, we used accuracy and the Brier-score. In the room occupancy dataset, we saw that S-RAD would in-sample consistently outperform the discretizer based on the median, using both the accuracy and the Brier-score. In the test sets, however, the median performed better than S-RAD, but this difference was never significant.

It should be noted, that the difference between S-RAD and the Median was much smaller in the training set than in the test set. The fact that the latter would not result in any significant difference can largely be explained by the sizable variance, as discussed in the previous chapter. While these results suffice in theory to verify our theory for this dataset, we should also quickly add that the out-of-sample results for S-RAD were far from ideal.

We suspect that this unexpected overperformance of the median discretizer is caused by the same underlying reason for why the variance in results tends to be high: the train sets might not fully represent the test sets, and the test sets can contain only a small number of actual unique datapoints. In those situations, using the median discretizer can be beneficial, since it assumes the least about the dataset and is least informative. This lack of information helps it in out-of-sample tests because it is not pushed towards false conclusions. S-RAD, on the other hand, uses a more complex model, which helps it achieve rather good in-sample scores, but due to the additional information, suffers out-of-sample.

To verify this assumption, we checked the test data and saw that, indeed, if we drop *humidity* and *light*, in the worse case only about half the values in the test set are unique. Discretizing using any approach would reduce this number even further. What also stood out during further investigation, was that in the room occupancy dataset, some test sets would contain only a single class. This could be a result of having to sample in blocks. Given that the median discretizer adds the least amount of information, it tends to most resemble the majority class

predictor, which in such test cases would be very beneficial. Such scores do provide a skewed view of overall performance, however. S-RAD might perform less well in these cases, but might be better at distinguishing between class labels, thereby being more explainable.

Switching to our second hypothesis provides us with some more information. S-RAD is never consistently outperformed by any alternative discretizer. Only one rather powerful model is capable of achieving significantly better accuracy in a single case, but nothing else. This observation, together with the previous hypothesis, helps us to conclude that S-RAD is at least a stable choice. When it has a lot of data with varying class labels, it will perform significantly better than the median discretizer. When the test sets are rather uniform, this advantage disappears but does not result in a significant disadvantage. S-RAD can therefore in the best case outperform the simplest discretizer, while also never being beaten by any alternative method. At least for the room occupancy dataset, this means that S-RAD would be a solid choice for automating discretization, though further improvements can be made, as we describe in Chapter 10.

9.2 Bike Counters

In the single bike counters datasets, we saw little differences between the different discretization methods. While S-RAD was able to outperform the median for the *snow* variable, based on accuracy, the differences were relatively minor or insignificant. The median discretizer was also never significantly better than S-RAD, thereby validating our first hypothesis for both accuracy and the Brier-score.

Our second hypothesis also holds for this dataset: there is no discretization method capable of consistently outperforming S-RAD. In Section 7.2.1 we saw that the relationship between *temperature* and *amount of bikers* is stronger than that between *snow* and *amount of bikers*. We see that this stronger relationship helps relatively simple discretization models to gain a better performance than S-RAD. For the *snow* variable with its weaker relationship, the roles are reversed. Here the additional power of S-RAD starts to help, and there is no longer any method that is capable of significantly outperforming it. This follows fairly closely what we have seen before: when there is a small amount of information to learn from, or there are strong relationships, the less complex discretizers outperforms S-RAD in some measures. When the situation grows more complex, S-RAD is better able to adapt.

For the combined bike counters dataset, the same conclusions hold. The S-RAD discretizer is consistently on par with the median for both the accuracy and the Brier-score. It also does not get consistently outperformed by any other model. As for this dataset, both our hypotheses stand.

9.3 Synthetic

The synthetic dataset is perhaps one of the more interesting ones, as it allows us to avoid the problems in testing methods we have seen so far. We are guaranteed that all test-sets contain many change points (due to our definition using the sine with a relatively short periodicity), as well as many data points. Class labels are fairly equally distributed, with a ratio of about 11:9. Under these conditions, we see S-RAD flourish: it is both significantly better than the median in both the test sets and train sets, for the accuracy as well as the Brier-score. This validates our hypothesis for the synthetic dataset. Also, not only is there no other discretization method that

significantly outperforms it, S-RAD is consistently the top-performer together with EFD-10, meaning that it has fully adapted itself to the dataset.

9.4 Summary

To summarize, our hypotheses hold for the datasets. S-RAD is a reliable option for automatic discretization: it consistently either outperforms or achieves the same scores as the discretization option that is regularly used when we have no prior knowledge about the distribution of the data. Sometimes choosing an appropriate discretization option by hand can be advantageous, but the specific model depends on the data so that prior knowledge would be required. S-RAD does not require this. Looking at both the accuracy and the Brier-score, S-RAD can learn models that classify well and assign accurate probabilities to our classifications.

Chapter 10

Further Research

Thus far we have seen that S-RAD is a solid choice for automatic discretization because it improves on the current strategy of discretizing based on the median while never being consistently outperformed by any other method that we have tested. While this is an important first step, we think that there are further improvements to be made. First, S-RAD is still rather rudimentary: it can only select entire discretization methods. Here we discuss a more sophisticated alternative that can fuse existing methods to potentially provide an even better discretization approach, which can also consistently outperform other methods. We also describe a method based on latent variables that would be more theoretically elegant and perhaps better at creating a good explainable model. Finally, we suggest a direction for how the work presented here can be made more generally applicable.

10.1 Fusion-based Reduced Automatic Discretization (F-RAD)

One area of further research that would be worth exploring would be to create an alternative to the rather naive S-RAD discretization approach. While S-RAD selects from a group of entire discretization methods, we think it would be interesting to extract the *boundaries* that existing methods produce and select the best from those, using the binary discretization nodes as described in Chapter 5. In that way, it would be able to *fuse* the best of all the discretization algorithms together. It would be especially interesting to see how the number of allowed parents in the graph would influence the accuracy in such a situation. Having many parents could again lead to overfitting and a high computational complexity (as in FAD), while a too low amount might not produce a good accuracy. In any case, the method as a whole would have more power to fit the underlying distribution and have more flexibility and therefore can improve on the work we have done on S-RAD.

10.2 Using latent variables

Much of our time in the chapters on automatic discretization has been spent on creating the right constraints on graph structures to learn a good discretization strategy. We find that discretization nodes will often find relationships to completely different parts of the graph (either by chance or by real correlation) or amongst another. We have also had to introduce the notion of a selection set, which aims to find the discretization method that helps most with classification while finding a graph that fits well with the data. While there is certainly value to

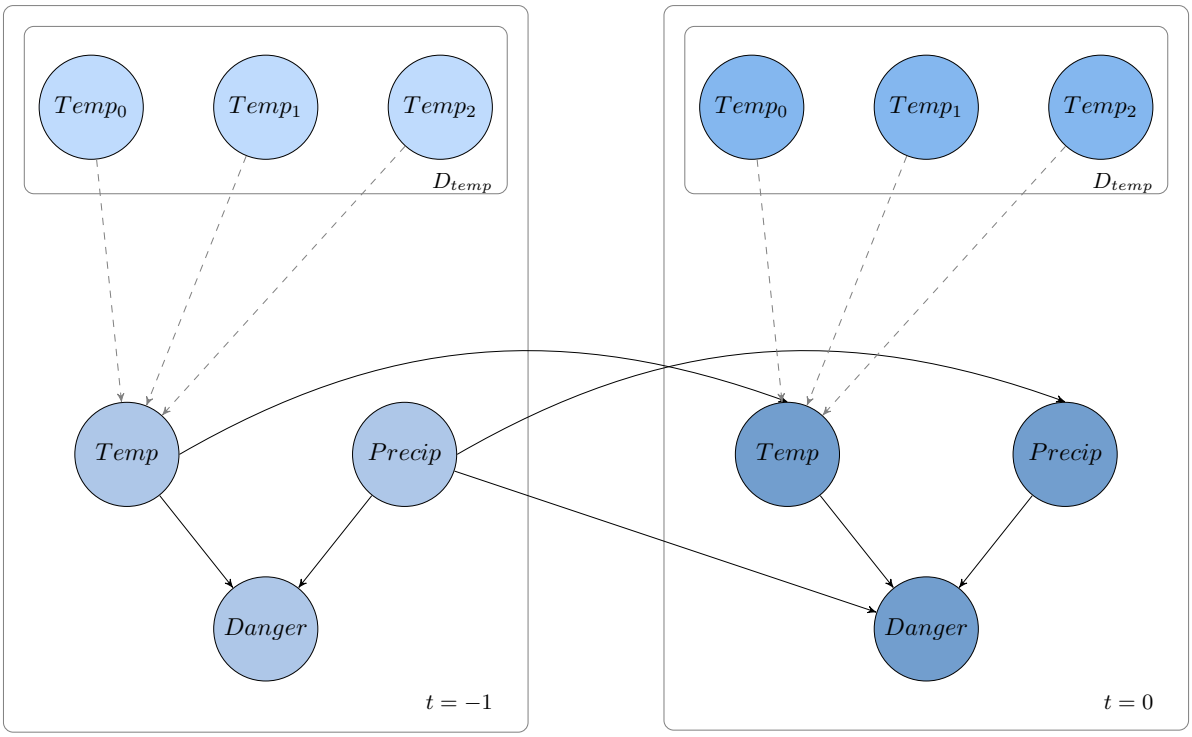


Figure 10.1: Example of a C-DBN with $Temp$ as latent variable needing to be discretized. The discretization sets of $Temp$ are isolated from the rest of the nodes. The structure learner can select one from the set to add information.

this approach, it would be interesting to model the variable that needs to be discretized directly. This could be done with the use of latent variables: nodes in the graph that have no data to learn from.

In this alternative approach, each node that needs to be discretized becomes a latent variable. The latent variable behaves like every other probabilistic variable in the C-DBN, but with the exception that it can select additional parents from a "discretization group". This group contains the various discretization methods (in S-RAD) or discretization borders (in F-RAD) that we want to investigate. This is shown in Figure 10.1.

By creating the latent variable, we have isolated the discretization nodes from the rest of the network, thereby limiting their influence to only modeling the variable that we want. This isolation does not only provide a cleaner theoretical approach but also makes it easier to implement. For our S-RAD implementation, we would no longer need to create constraints that make the hidden node choose for zero or one node of each discretization method (which is not trivial in GOBNILP's constraint language). Instead, we could add an ordinary constraint on the number of parents a latent variable can have from its discretization set.

Since we would be modeling the variables to be discretized directly, instead of via the hidden variable, we can expect our results to lose accuracy in the discretization tasks. On the other hand, we expect the approach with latent variables to produce a more faithful representation at the discretization level, thereby possibly increasing out-of-sample performance. We would be very interested to see whether these intuitions are correct, or if there are other factors at play.

What should be noted about this approach though is that using latent variables does add an extra layer of complexity. For every latent variable, we require an additional loop overall structure learning possibilities and parameter learning options [28, p. 917]. This might make

finding an exact solution impossible, forcing us to optimization methods such as Expectation Maximization [28, p. 921]. We would be interested to see the real-world implications on model performance of such a switch.

10.3 DBN-transformation with multiple measurements

A limiting factor of this project was that there was a relatively small number of datasets openly available, that were both temporal and required classification, even though this problem might occur quite frequently in the real-world. What made it even harder to find good data sets was that not all temporal classification datasets were supported, due to our implementation. This was the case when there are multiple measurement values for each point in time. An example of this occurred in an airline database, in which the status of many planes was recorded. Because we used shifting for transforming dynamic datasets into regular ones, this caused problems in our learning approach. A naive solution to this problem would be to shift groups of rows rather than individual rows. That approach has some limiting assumptions: both the order and number of measurements at each point in time have to be static. Creating a robust solution to this problem would help make our method more widely applicable.

Chapter 11

Conclusion

In this project, we have aimed to create a method of automatically learning an explainable method for classification tasks in time series. This is an area of research which, in our opinion, was underserved. Since the classification of time series has many practical applications and public opinion trends towards preferring explainable models, we believe that this was a problem worth exploring. Since classifying time-series in an explainable way is a rather new area, it was our goal to provide the first step towards a fitting solution. To this extent, we have laid some important groundwork in this project.

First, we have formulated a transformation from DBNs to regular BNs and back, so existing tools can be used to learn both the structure and the parameters of these networks to create explainable and dynamic models. We have created both a theoretical framework, as well as implementing it in code. Second, we wanted to improve usability by shifting the standard machine learning task of discretization towards the automatic structure learning framework. We first showed an approach that, while powerful, is also computationally infeasible. We attempted to solve this using existing discretization methods as heuristics, in an approach we called S-RAD. While we have seen some good first results using this method, we also proposed ways of continuing research towards good automatic discretization that can potentially be even more flexible.

In the process of doing this research, we have also worked on adjacent issues. We introduced a new way of validating discretization and classification algorithms in time-series, based on earlier work in Econometrics using k, v-fold cross-validation and the partial autocorrelation function. We also explored which datasets can currently be used for classification of time-series. We open-sourced most of our implementations (see Appendix B, C, and D) to help speed-up further research and adoption, as well as helping to fix problems in existing projects.

To conclude, we have shown classifying time-series in an explainable way to be a solvable problem. We have created methods, show significant potential. We are excited to see further research in this area.

Chapter 12

Acknowledgments

This thesis was written for the Computing Science Master for Utrecht University in 2019. The author was financially supported by de Volksbank N.V. to do this research. The author also received utilities required for this project, such as AWS EC2 server time. De Volksbank did not assert any constraints on the research, other than that internal data would not be used.

Bibliography

[1] Occupancy Detection Dataset. https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+. Accessed: 15-07-2019.

[2] Ottawa Bike Counters - Time-series of 15 counter locations in Ottawa Ontario Canada over 8 years. https://www.kaggle.com/m7homson/ottawa-bike-counters. Accessed: 16-05-2019.

[3] USA Environmental Protection Agency. Air Quality Index (AQI) Basics.

[4] Ankur Ankan. PGMPY. http://pgmpy.org/. Accessed: 30-06-2019.

[5] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 5^{th} edition, 2015.

[6] Glenn W. Brier. Verification of forecasts expressed in terms of probability. *Monthly Weather Review*, 78(1):1–3, 1950.

[7] Cassio P de Campos and Qiang Ji. Efficient structure learning of Bayesian networks using constraints. *Journal of Machine Learning Research*, 12(Mar):663–689, 2011.

[8] Luis M Candanedo and Véronique Feldheim. Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. *Energy and Buildings*, 112:28–39, 2016.

[9] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[10] Juan Chen, Ping-An Zhong, Ru An, Feilin Zhu, and Bin Xu. Risk analysis for real-time flood control operation of a multi-reservoir system using a dynamic Bayesian network. *Environmental Modelling & Software*, 111:409 – 420, 2019.

[11] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, 2015.

[12] James Cussens, Matti Järvisalo, Janne H Korhonen, and Mark Bartlett. Bayesian network structure learning with integer programming: Polytopes, facets and complexity. *Journal of Artificial Intelligence Research*, 58:185–229, 2017.

[13] Cassio P De Campos and Qiang Ji. Improving Bayesian network parameter learning using constraints. In *Proceedings of the 19th International Conference on Pattern Recognition*, pages 1–4. IEEE, 2008.

[14] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *Machine Learning Proceedings 1995*, pages 194–202. Elsevier, 1995.

[15] Dheeru Dua and Casey Graff. UCI Machine Learning Repository, 2017.

[16] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & security*, 28(1-2):18–28, 2009.

[17] Zoubin Ghahramani and Michael I Jordan. Factorial hidden Markov models. *Machine Learning*, 29:245–275, 1997.

[18] David C Hamilton and Donald G Watts. Interpreting partial autocorrelation functions of seasonal time series models. *Biometrika*, 65(1):135–140, 1978.

[19] Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 291–301. IEEE, 2002.

[20] Frank E Harrell Jr, Kerry L Lee, and Daniel B Mark. Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors. *Statistics in medicine*, 15(4):361–387, 1996.

[21] David Heckerman. A tutorial on learning with Bayesian networks. In *Innovations in Bayesian networks*, pages 33–82. Springer, 2008.

[22] David Heckerman, Dan Geiger, and David M Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243, 1995.

[23] Aude Hofleitner, Ryan Herring, Pieter Abbeel, and Alexandre Bayen. Learning the dynamics of arterial traffic from probe data using a dynamic Bayesian network. *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1679–1693, 2012.

[24] Koninklijk Nederlands Meteorolgisch Instituut. KNMI waarschuwingen Nederland. 2015. `https://cdn.knmi.nl/system/downloads/files/000/000/017/original/WaarschuwingenKNMI_algemeen_versiejuni_2015.pdf`.

[25] Tommi Jaakkola, David Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 358–365, 2010.

[26] George H John and Pat Langley. Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[27] Nima Khakzad. Modeling wildfire spread in wildland-industrial interfaces using dynamic Bayesian network. *Reliability Engineering & System Safety*, 189:165–176, 2019.

[28] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[29] Kevin B Korb and Ann E Nicholson. *Bayesian Artificial Intelligence*. CRC press, 2010.

[30] Huan Liu, Farhad Hussain, Chew Lim Tan, and Manoranjan Dash. Discretization: An enabling technique. *Data mining and knowledge discovery*, 6(4):393–423, 2002.

[31] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. In *Proceedings of the 23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 89–94. Presses universitaires de Louvain, 2015.

[32] Kevin Patrick Murphy and Stuart Russell. Dynamic Bayesian networks: representation, inference and learning. 2002.

[33] Richard E Neapolitan. *Learning Bayesian networks*. Pearson Prentice Hall Upper Saddle River, NJ, 2004.

[34] Vladimir Pavlovic, James M Rehg, Tat-Jen Cham, and Kevin P Murphy. A dynamic Bayesian network approach to figure tracking using learned dynamic models. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 1, pages 94–101. IEEE, 1999.

[35] AN Pettitt. A non-parametric approach to the change-point problem. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 28(2):126–135, 1979.

[36] Ole Raaschou-Nielsen, Zorana J Andersen, Rob Beelen, Evangelia Samoli, Massimo Stafoggia, Gudrun Weinmayr, Barbara Hoffmann, Paul Fischer, Mark J Nieuwenhuijsen, Bert Brunekreef, et al. Air pollution and lung cancer incidence in 17 European cohorts: prospective analyses from the European study of Cohorts for Air Pollution Effects (ESCAPE). *The lancet oncology*, 14(9):813–822, 2013.

[37] Lawrence R Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[38] Jeff Racine. Consistent cross-validatory model-selection for dependent data: hv-block cross-validation. *Journal of Econometrics*, 99(1):39 – 61, 2000.

[39] Silja Renooij. Probability elicitation for belief networks: issues to consider. *The Knowledge Engineering Review*, 16(3):255–269, 2001.

[40] Mauro Scanagatta, Cassio P de Campos, Giorgio Corani, and Marco Zaffalon. Learning Bayesian networks with thousands of variables. In *Advances in neural information processing systems*, pages 1864–1872, 2015.

[41] Tomi Silander, Petri Kontkanen, and Petri Myllymäki. On sensitivity of the MAP Bayesian network structure to the equivalent sample size parameter. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pages 360–367. AUAI Press, 2007.

[42] Allan Tucker, Veronica Vinciotti, Peter AC't Hoen, and Xiaohui Liu. Bayesian network classifiers for time-series microarray data. In *International Symposium on Intelligent Data Analysis*, pages 475–485. Springer, 2005.

[43] Xianguo Wu, Huitao Liu, Limao Zhang, Miroslaw J. Skibniewski, Qianli Deng, and Jiaying Teng. A dynamic Bayesian network based approach to safety decision support in tunnel construction. *Reliability Engineering & System Safety*, 134:157 – 168, 2015.

[44] Ying Yang and Geoffrey I Webb. A comparative study of discretization methods for naive-Bayes classifiers. In *Proceedings of PKAW*, volume 2002, 2002.

Appendix A

Dataset Exploration

Throughout this project, we have considered many datasets for usage as an example or for testing. In this part of the appendix, we will describe the various datasets we have explored and give reasons for why they have or have not been used.

A.1 KNMI-Weather Alarms

The very first dataset which we have used for many months in the course of this project has been a weather alarms dataset based on data from the Royal Dutch Institute for Meteorology (See: `http://projects.knmi.nl/klimatologie/uurgegevens/selectie.cgi`). This dataset contains a large number of variables including wind speed, wind direction, temperature, amount of precipitation, fog, and more measured once every hour for more than 20 years. With such a large amount of data, we hoped to avoid a common problem in machine learning: a severe lack of data to learn from.

To make the study interesting, we wanted to couple this data with information about when weather alarms had been announced. These weather alarms indicate several levels of potentially dangerous weather to the general public. We thought this would be an excellent showcase for C-DBNs as it could help predict dangerous situations as well as explaining why it would do so. In addition to that, we would be able to check this explanation as various levels of alarms are generated based on static thresholds, while others are only created based on expert opinion [24].

To achieve the dataset we required for learning such a model, we combined the previously mentioned KNMI dataset with publically available information on "Red" and "Orange" weather alarms. After much testing, we found that we were unable to outperform even a majority class prediction model on the dataset we constructed, even though the graphs we constructed seemed to indicate that the C-DBN had indeed learned quite some valuable relationships, as shown in Figure A.1.

The big problem of this dataset and the reason why we opted not to use it is that it lacks some essential classifications. While there was openly available data for the "Red" and "Orange" weather alerts, "Yellow" weather alerts were not included, while the difference between "Yellow" and "Orange" can in many real-world instances be somewhat arbitrary. For this reason, we would often classify situations as "Orange" when they would not have any alert in the dataset, while in reality a "Yellow" alert had been sent out. In one instance when we "wrongly" classified a day as "orange" , there had been so much snow that traffic had almost come to a complete

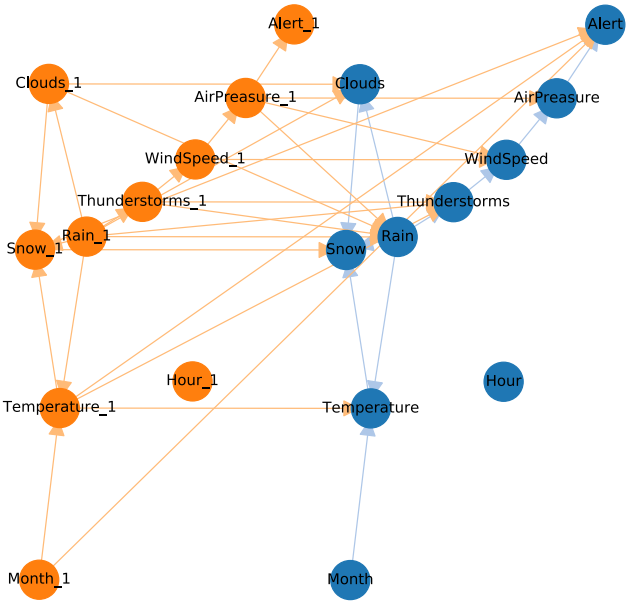


Figure A.1: Visual representation of the learned C-DBN for the KNMI Weather Alarm Dataset.

stop in the Netherlands. While this certainly shows the potential of our method, we could not accurately quantify its performance without having more data about "yellow" alerts. We tried to obtain these separately, but sadly to no avail. Therefore we had to leave this dataset out of the main body of this paper.

A.2 Fraud Detection

Another dataset that we explored is called *Fraud Detection* (see: `https://www.kaggle.com/ntnu-testimon/paysim1`), which contains many transactions. The classification task here is to find which transactions are fraudulent. This dataset is good at illustrating the limitations of our approach. While at first, it might seem to fit our models fairly well, since it is a classification problem for dynamic data, it cannot work with the C-DBN method we have proposed. That is because its values are neither temporal nor measured at set intervals. Therefore, we cannot use the unrolling strategy and the conventional way of predicting values. For that reason, we have not included the database in our project.

A.3 Absenteeism at Work

An interesting dataset which we explored is called *Absenteeism at Work* and is part of the UCI Machine Learning Repository (see: `https://archive.ics.uci.edu/ml/datasets/Absenteeism+at+work`). It is a set of 740 rows and 21 variables. The variables provide a wide array of information about cases in which employees had been absent. The classification task would involve predicting one of several reasons for why an employee had not come to work. While the UCI classified this as a time-series, it contained neither a time component nor a sequence

of measurements. We therefore quickly discovered that this dataset too could not be used for C-DBNs. Regular BNs would in this case probably suffice.

A.4 PM2.5

PM2.5, refers to fine particulate matter that causes increased levels of respiratory problems in humans [36]. The UCI Machine Learning Repository contains a data set in which levels of these particulates are recorded in Beijing over a period of time (see: `https://archive.ics.uci.edu/ml/datasets/Beijing+PM2.5+Data`). We wanted to create a C-DBN learner which would be capable of predicting elevated levels of PM2.5 that would be hazardous for specific groups of people in Beijing. This could have real-world potential: if we can not only warn people of risk ahead of time but also show the most likely contributing factors, residents of the city can make more informed decisions for their health.

The database itself was not labeled yet: it did not contain any information on whether a level of fine particulate matter would be dangerous. To label the data, we used the US-EPA Air Quality Index [3] and chose either the *Very Unhealthy* or *Hazardous* values. After experimenting for a while, we found out that the accuracy of our C-DBN on the dataset was, again, very poor compared to majority class prediction. After investigation, we found out that creating the classification values from the AQI caused the problem. There would be many cases around the edges, which would fall just the wrong way because there was an arbitrary strict threshold involved. Since this was a problem with the dataset itself, we chose not to use it further.

For more research, there would probably be an alternative approach here: by not looking at accuracy directly, but at the Brier-score, which also describes its predictive performance (not only *was it good enough* but *how close was our guess*), we could more clearly see how much we would have learned. Since for any real-world situation such a static threshold is not likely to exist, this would probably be a logical approach.

Appendix B

Constraining Parent Set Generation beyond regular GOBNILP

There is only one type of constraints on the DAG that GOBNILP can currently use for structure learning: those of the form $A \leftarrow B$, for $A \in Pa(B)$, and its negation $\neg A \leftarrow B$. In GOBNILPs manual and examples references are made to its capability of using conditional independencies. However, at the moment of writing, this had not yet been implemented. Therefore it is not directly possible to make statements of the kind $A \leftarrow B$ if $\neg C \leftarrow B$. Since this is required for S-RAD (as described in Section 6.4), we had to find a workaround.

Another problem we faced is that GOBNILP only enforces constraints after it has already generated the parent-sets and scores for the combinations. Since we use many constraints that bring down the number of combinations to be checked dramatically, the regular GOBNILP approach causes many unnecessary computations.

Both of these problems can be solved by splitting the structure learning problem in two parts: parent-set generation, and finding a structure that maximizes the parent-set scores without creating an invalid graph. We accomplished this by extracting and slightly altering the GOBNILP parent-set generator. Our altered version specifically checks constraints beforehand, thereby solving the second problem. To solve our first problem, we filtered out parent-sets that contained illegal combinations given the constraints in Section 6.3. We then let GOBNILP continue with the parent-set combinations which we created.

For GOBNILP to read our generated parent set combinations, we had to follow a strict format called JKL. This was first introduced by Jaakkola [25]. To be able to read the generated parent sets that were created by our scoring algorithm, filter them, and write them back to the format GOBNILP required, we created a JKL-Serialization package for Python. This has been made openly available on GitHub (see: `https://github.com/daanknoope/jkl-serialization`) and is part of the Python package repository PyPi (`https://pypi.org/project/jkl-serialization/`).

Appendix C

PGMPY - Bugfixes and Workarounds

In working with the PGMPY library for parameter learning of Bayesian networks in Python, we have experienced several issues. For some of these we have been able to create fixes, for others we used a workaround. For the sake of reproducibility, we will describe these adaptations in this appendix.

C.1 Mapping of values to list indices in CPDs

One of the major problems in PGMPY was that it assumed (without explicitly stating this) that all values supplied as evidence in variable elimination could be used as list indices for the CPDs. If we have a variable v for example, that takes values 0 and 1, then we could query the CPD by looking at place 0 or 1 in the list of probabilities for v. This, however, does not work when values are not zero-indexed, when some values are larger than the number of possible values, or when not every list index corresponds to a value. In those cases, we require a mapping between values and list indices. An example of this is shown in Listing C.1.

```
from pgmpy.inference import VariableElimination
from pgmpy.models import BayesianModel
import numpy as np
import pandas as pd

# Create a sample dataset with values in [2,3]
values = pd.DataFrame(np.random.randint(low=2, high=4, size=(1000, 5)),
                      columns=['A', 'B', 'C', 'D', 'E'])
model = BayesianModel([('A', 'B'), ('C', 'B'), ('C', 'D'), ('B', 'E')])
model.fit(values)

inference = VariableElimination(model)
phi_query = inference.map_query(['A'], evidence={'B':3})
print(phi_query)

## Expected: MAP value of A given B=3
## Result: Error (index 3 is out of bounds)
```

Listing C.1: Code sample for showing wrong index for evidence in inference

In the best case, this bug produces an `index is out of bounds` error, but in the worst case, it leads to erroneous results. If we do have zero-indexed variables that have the exact correct length but are sorted differently, we will get results that correspond to different values.

Since PGMPY is an open-source project, we reported an issue (see: `https://github.com/pgmpy/pgmpy/issues/1071`) that the author fixed during this project. In some earlier parts of the project, when the issue had not been solved, we manually changed all variables so the indexes would correspond to these values. Some parts of the code might still refer to this, as it is not affected by the fix and will continue running correctly.

C.2 Progress-bar running out of RAM

Another problem that we found during the project was that a progress bar had been introduced for Variable Elimination. While this can be a nice addition when a small number of queries are being done, the implementation has made its addition problematic for our research. When we wanted to query many different variable configurations over various graphs for our testing, we found that either the continuous generation of these objects, or their continuous writing to the computer's output, would gradually cause the computer to use up all its RAM. This approached a point where every experiment would crash because the computer would run both out of RAM (40GB) and out of SWAP (16GB). We notified the author of this behavior, but at the time of writing, no official fix has been published. As a workaround, we have removed the progress bar from PGMPY.

The changes for this are available both on `https://github.com/daanknoope/pgmpy` and are shown in Listing C.2 as a git patch.

```
From c91ef471fe44cc8a05299accae5c7b8e85a0b677 Mon Sep 17 00:00:00 2001
From: Daan Knoope <daan@knoope.dev>
Date: Wed, 3 Jul 2019 00:43:17 +0200
Subject: [PATCH] Remove progress bar

The progress bar has been removed from the variable elimination process since it causes Jupyter to crash
      when looping through networks.
---
 pgmpy/inference/EliminationOrder.py | 20 ++++++++------------
 pgmpy/inference/ExactInference.py   | 13 ++++---------
 2 files changed, 10 insertions(+), 23 deletions(-)

diff --git a/pgmpy/inference/EliminationOrder.py b/pgmpy/inference/EliminationOrder.py
index 3d2fbef..46be619 100644
--- a/pgmpy/inference/EliminationOrder.py
+++ b/pgmpy/inference/EliminationOrder.py
@@ -1,6 +1,5 @@
 from abc import abstractmethod
 from itertools import combinations
-from tqdm import tqdm

 import numpy as np

@@ -88,17 +87,14 @@ class BaseEliminationOrder:

         ordering = []

-        with tqdm(total=len(nodes)) as pbar:
-            pbar.set_description("Finding Elimination Order: ")
```

```
-            while nodes:
-                scores = {node: self.cost(node) for node in nodes}
-                min_score_node = min(scores, key=scores.get)
-                ordering.append(min_score_node)
-                nodes.remove(min_score_node)
-                self.bayesian_model.remove_node(min_score_node)
-                self.moralized_model.remove_node(min_score_node)
-
-                pbar.update(1)
+        while nodes:
+            scores = {node: self.cost(node) for node in nodes}
+            min_score_node = min(scores, key=scores.get)
+            ordering.append(min_score_node)
+            nodes.remove(min_score_node)
+            self.bayesian_model.remove_node(min_score_node)
+            self.moralized_model.remove_node(min_score_node)
+
         return ordering

     def fill_in_edges(self, node):
diff --git a/pgmpy/inference/ExactInference.py b/pgmpy/inference/ExactInference.py
index 34e8a7e..3936c8e 100644
--- a/pgmpy/inference/ExactInference.py
+++ b/pgmpy/inference/ExactInference.py
@@ -4,7 +4,6 @@ import itertools

 import networkx as nx
 import numpy as np
-from tqdm import tqdm
 from pgmpy.extern.six.moves import filter, range

 from pgmpy.extern.six import string_types
@@ -159,16 +158,8 @@ class VariableElimination(Inference):
         )

         # Step 3: Run variable elimination
-        pbar = tqdm(elimination_order)
-        for var in pbar:
-            pbar.set_description("Eliminating: {var}".format(var=var))
-            # Removing all the factors containing the variables which are
-            # eliminated (as all the factors should be considered only once)
-            factors = [
-                factor
-                for factor in working_factors[var]
-                if not set(factor.variables).intersection(eliminated_variables)
-            ]
+        for var in elimination_order:
+            factors = [factor for factor in working_factors[var] if not set(factor.variables).
    intersection(eliminated_variables)]
             phi = factor_product(*factors)
             phi = getattr(phi, operation)([var], inplace=False)
             del working_factors[var]
--
2.17.1
```

Listing C.2: Git patch for removing progress bar to work around memory leak problem

C.3 Torch vs PyTorch

A minor problem that is worth noting while installing PGMPY is that the requirement files are broken at the time of writing when using Python's *pip* package installer, which is required for installing all of the packages created for this project. The requirement files (`requirements.txt` and `requirements-dev.txt`) are written for Anaconda's package manager, in which PyTorch is referred to as `PyTorch`, while in pip it is called `torch`. Changing these names causes pip to read the requirement files correctly, otherwise not all the libraries for building PGMPY will be installed.

Note that this is only relevant when installing PGMPY from source, not when downloading it from a package repository. For this project, however, we have been working on the latest development version with the patch mentioned in Section C.2.

C.4 State names not being passed

In previous versions of PGMPY, fitting parameters on a graph would cause variable names in the CPDs to change. Where the probability for Variable V with value 0 would be referred to as $A(0)$ before learning, this would change to A_0 afterward. Since we used the underscore already in our program for denoting time frames, this inconsistency caused a range of problems. To solve this, we have fixed the underlying problem, and the fix has been integrated into PGMPY. See `https://github.com/pgmpy/pgmpy/issues/1062` for more details.

Appendix D

Experimentation environment

Our results were gathered on an Ubuntu 18.04.2 (4.15.0-54-generic) installation, using Python 3.7.3 on Anaconda 4.6.11. For GOBNILP version 1.6.3 was used, as compiled with SCIP version 3.2.0. Since the compilation process for GOBNILP is not entirely trivial, we created a Docker container to make the process reproducible. The container also includes a recipe for how GOBNILP can be installed on any Ubuntu 18.04 installation. We have made it openly available on GitHub: `https://github.com/daanknoope/gobnilp-container`.

We used a development branch of PGMPY, version `c34ccd7be1d`, as released on Friday the 21st of June, 2019 together with a patch which we describe in more detail in Appendix C.2. The exact version of all the packages we used is shown in the list below.

- asn1crypto==0.24.0
- atomicwrites==1.3.0
- attrs==18.2.0
- backcall==0.1.0
- bleach==3.1.0
- certifi==2019.6.16
- cffi==1.12.2
- chardet==3.0.4
- coverage==4.5.2
- cryptography==2.6.1
- cycler==0.10.0
- DBN-learner==0.1
- decorator==4.3.2
- Discretizer==0.1
- entrypoints==0.3
- gobnilp==0.1
- idna==2.8
- ipykernel==5.1.0
- ipython==7.2.0
- ipython-genutils==0.2.0
- jedi==0.13.2
- Jinja2==2.10
- jkl-serialization==0.0.1
- joblib==0.13.2
- jsonschema==3.0.1
- jupyter-client==5.2.4
- jupyter-core==4.4.0
- kiwisolver==1.0.1
- kv-block-cross-validation==0.1
- MarkupSafe==1.1.0
- matplotlib==3.0.2
- mdlp-discretization==0.3.2
- mistune==0.8.4
- mkl-fft==1.0.10
- mkl-random==1.0.2
- mock==2.0.0
- more-itertools==5.0.0
- nbconvert==5.3.1
- nbformat==4.4.0
- network2tikz==0.1.8
- networkx==2.3
- nose==1.3.7
- numpy==1.16.1
- palettable==3.1.1
- pandas==0.24.1
- pandocfilters==1.4.2
- parso==0.3.1
- patsy==0.5.1
- pbr==5.1.3
- pexpect==4.6.0

- pgmpy==0.1.8.dev31
- pickleshare==0.7.5
- pluggy==0.8.1
- prompt-toolkit==2.0.9
- ptyprocess==0.6.0
- py==1.7.0
- pycparser==2.19
- Pygments==2.3.1
- pygraphviz==1.5
- pyOpenSSL==19.0.0
- pyparsing==2.3.1
- pyrsistent==0.14.11
- PySocks==1.6.8
- pytest==4.2.1
- pytest-cov==2.6.1
- python-dateutil==2.7.5
- pytz==2018.9
- pyzmq==17.1.2
- requests==2.21.0
- scikit-learn==0.20.3
- scipy==1.2.0
- seaborn==0.9.0
- Send2Trash==1.5.0
- six==1.12.0
- sklearn==0.0
- statsmodels==0.9.0
- terminado==0.8.1
- testpath==0.4.2
- torch==1.1.0
- tornado==5.1.1
- tqdm==4.31.1
- traitlets==4.3.2
- urllib3==1.24.1
- wcwidth==0.1.7
- webencodings==0.5.1
- wrapt==1.11.1

Appendix E

Additional Results

E.1 Bike Single

				Accuracy		Brier-Score	
				mean	std	mean	std
Dataset	Variable	Method	Bins				
Test	Snow	EFD	2.0	0.754662	0.154809	0.131940	0.075656
			3.0	0.754662	0.154809	0.131940	0.075656
			10.0	0.766881	0.133390	0.129740	0.081518
		EWD	2.0	0.740193	0.155126	0.133700	0.074466
			3.0	0.742122	0.151999	0.130583	0.078160
			10.0	0.755949	0.132458	0.135877	0.083468
		IQR	0.0	0.766559	0.133596	0.125035	0.075744
		Median	0.0	0.768167	0.137597	0.122636	0.068519
		SRAD	0.0	0.767846	0.134484	0.123121	0.071688
	Temp	EFD	2.0	0.757878	0.147840	0.130835	0.076112
			3.0	0.750804	0.153589	0.127920	0.074619
			10.0	0.754662	0.154809	0.131940	0.075656
		EWD	2.0	0.770096	0.138029	0.126852	0.073639
			3.0	0.759164	0.150363	0.128773	0.076724
			10.0	0.755949	0.150874	0.131164	0.073258
		IQR	0.0	0.748553	0.152369	0.129512	0.075299
		Median	0.0	0.757878	0.147840	0.130835	0.076112
		SRAD	0.0	0.760450	0.150293	0.129124	0.076676
Train	Snow	EFD	2.0	0.756771	0.020526	0.118321	0.008697
			3.0	0.756771	0.020526	0.118321	0.008697
			10.0	0.775041	0.015883	0.108798	0.007277
		EWD	2.0	0.758080	0.019854	0.113270	0.007359
			3.0	0.760476	0.019990	0.111329	0.008392
			10.0	0.777142	0.018458	0.109285	0.008379
		IQR	0.0	0.783419	0.018876	0.107302	0.008377
		Median	0.0	0.779933	0.019377	0.108518	0.008378
		SRAD	0.0	0.782331	0.019066	0.107769	0.008465
	Temp	EFD	2.0	0.764505	0.019020	0.114983	0.008667
			3.0	0.769447	0.018987	0.111939	0.008461
			10.0	0.756771	0.020526	0.118321	0.008697
		EWD	2.0	0.770098	0.018571	0.113776	0.008472
			3.0	0.765231	0.019612	0.112536	0.008839
			10.0	0.757958	0.024017	0.117307	0.011749
		IQR	0.0	0.765049	0.019811	0.113706	0.008898
		Median	0.0	0.764505	0.019020	0.114983	0.008667
		SRAD	0.0	0.765049	0.019645	0.112748	0.008926

Table E.1: Full scores of the bike counters single dataset ($n = 10$)

Dataset	Variable	Score	Score SRAD	Std. SRAD	Score Median	Std. Median	p
Train	Snow	Accuracy	0.782	0.019	0.780	0.019	0.006
Train	Snow	Brier-Score	0.108	0.008	0.109	0.008	0.150
Train	Temp	Accuracy	0.765	0.020	0.765	0.019	0.439
Train	Temp	Brier-Score	0.113	0.009	0.115	0.009	0.000
Test	Snow	Accuracy	0.768	0.134	0.768	0.138	0.899
Test	Snow	Brier-Score	0.123	0.072	0.123	0.069	0.696
Test	Temp	Accuracy	0.760	0.150	0.758	0.148	0.479
Test	Temp	Brier-Score	0.129	0.077	0.131	0.076	0.154

Table E.2: Comparison of results between S-RAD and Median for the bike counters dataset with a single variable, based on different scores and measured variables.

E.2 Bike Combined

				Accuracy		Brier-Score	
				mean	std	mean	std
Dataset	Variable	Method	Bins				
Test	Snow	EFD	2.0	0.753376	0.141276	0.126098	0.076870
			3.0	0.753376	0.141276	0.126098	0.076870
			10.0	0.764309	0.144389	0.128159	0.076202
		EWD	2.0	0.742122	0.142875	0.129313	0.076538
			3.0	0.748232	0.142144	0.129261	0.080348
			10.0	0.752412	0.143014	0.138344	0.089884
		IQR	0.0	0.767846	0.134082	0.123342	0.073580
		Median	0.0	0.769453	0.135880	0.124162	0.073665
		SRAD	0.0	0.769453	0.134101	0.123996	0.073443
	Temp	EFD	2.0	0.770096	0.131307	0.125164	0.068251
			3.0	0.766238	0.133043	0.129507	0.074956
			10.0	0.767203	0.134829	0.126773	0.067348
		EWD	2.0	0.772347	0.137842	0.133573	0.085481
			3.0	0.758521	0.130043	0.125916	0.072959
			10.0	0.767203	0.134829	0.127995	0.071115
		IQR	0.0	0.754019	0.132985	0.127222	0.071064
		Median	0.0	0.770096	0.131307	0.125164	0.068251
		SRAD	0.0	0.764952	0.130413	0.126428	0.073078
Train	Snow	EFD	2.0	0.771253	0.020071	0.109382	0.008975
			3.0	0.771253	0.020071	0.109382	0.008975
			10.0	0.774599	0.017811	0.110466	0.009263
		EWD	2.0	0.769581	0.020765	0.106238	0.008339
			3.0	0.768854	0.022016	0.106793	0.010420
			10.0	0.777070	0.018254	0.108982	0.009278
		IQR	0.0	0.780672	0.015763	0.107820	0.007954
		Median	0.0	0.779329	0.015557	0.108995	0.008036
		SRAD	0.0	0.780454	0.015862	0.108121	0.008054
	Temp	EFD	2.0	0.777576	0.017518	0.108310	0.008495
			3.0	0.777542	0.019111	0.108432	0.009512
			10.0	0.772527	0.018656	0.111686	0.008551
		EWD	2.0	0.780187	0.018343	0.108693	0.008238
			3.0	0.780154	0.017503	0.107134	0.008632
			10.0	0.772383	0.018237	0.111750	0.008363
		IQR	0.0	0.772273	0.017954	0.111594	0.008375
		Median	0.0	0.777576	0.017518	0.108310	0.008495
		SRAD	0.0	0.778337	0.018226	0.107729	0.008983

Table E.3: Full scores of the bike counters combined dataset ($n = 10$)

Dataset	Variable	Score	Score SRAD	Std. SRAD	Score Median	Std. Median	p
Train	Snow	Accuracy	0.780	0.016	0.779	0.016	0.042
Train	Snow	Brier-Score	0.108	0.008	0.109	0.008	0.002
Train	Temp	Accuracy	0.778	0.018	0.778	0.018	0.113
Train	Temp	Brier-Score	0.108	0.009	0.108	0.008	0.084
Test	Snow	Accuracy	0.769	0.134	0.769	0.136	1.000
Test	Snow	Brier-Score	0.124	0.073	0.124	0.074	0.739
Test	Temp	Accuracy	0.765	0.130	0.770	0.131	0.435
Test	Temp	Brier-Score	0.126	0.073	0.125	0.068	0.445

Table E.4: Comparison of results between S-RAD and Median for the bike counters dataset with multiple variables.

E.3 Room Occupancy

Dataset	Variable	Score	Score SRAD	Std. SRAD	Score Median	Std. Median	p
Train	CO2	Accuracy	0.991	0.003	0.987	0.003	0.006
Train	CO2	Brier-Score	0.007	0.002	0.010	0.002	0.008
Train	Light	Accuracy	0.991	0.003	0.985	0.003	0.000
Train	Light	Brier-Score	0.007	0.002	0.011	0.002	0.000
Train	Humidity	Accuracy	0.991	0.002	0.984	0.004	0.000
Train	Humidity	Brier-Score	0.008	0.002	0.012	0.003	0.000
Test	CO2	Accuracy	0.716	0.310	0.719	0.281	0.962
Test	Light	Accuracy	0.852	0.215	0.970	0.032	0.117
Test	Humidity	Accuracy	0.865	0.158	0.912	0.095	0.421

Table E.5: Comparison of results between S-RAD and Median, based on different scores and measured variables for the room occupancy dataset.

				Accuracy		Brier-Score	
				mean	std	mean	std
Dataset	Variable	Method	Bins				
Test	CO2	EFD	2.0	0.718573	0.281472	0.181723	0.046394
			3.0	0.772448	0.243944	0.174352	0.062412
			10.0	0.883887	0.139681	0.060301	0.029581
		EWD	2.0	0.879582	0.130344	0.145140	0.043147
			3.0	0.837515	0.213320	0.157821	0.056265
			10.0	0.826691	0.177482	0.132695	0.074018
		IQR	0.0	0.782042	0.202008	0.160270	0.062950
		Median	0.0	0.718573	0.281472	0.181723	0.046394
		SRAD	0.0	0.715990	0.310291	0.172752	0.018475
	Humidity	EFD	2.0	0.911685	0.094883	0.078922	0.051293
			3.0	0.901722	0.126651	0.136539	0.053467
			10.0	0.858180	0.164647	0.147472	0.062944
		EWD	2.0	0.929889	0.078449	0.108045	0.063823
			3.0	0.864207	0.157560	0.147891	0.042968
			10.0	0.843665	0.175850	0.156766	0.080912
		IQR	0.0	0.797786	0.229769	0.118565	0.054565
		Median	0.0	0.911685	0.094883	0.078922	0.051293
		SRAD	0.0	0.864699	0.157975	0.152738	0.039589
	Light	EFD	2.0	0.592866	0.311953	0.204216	0.020631
			3.0	0.946125	0.080709	0.046193	0.040531
			10.0	0.918573	0.118389	0.012795	NaN
		IQR	0.0	0.816851	0.261622	0.061263	0.073429
		Median	0.0	0.970111	0.031642	0.048991	0.043155
		SRAD	0.0	0.852030	0.215106	0.060286	0.093309
Train	CO2	EFD	2.0	0.986948	0.003063	0.009587	0.002153
			3.0	0.987741	0.005134	0.009130	0.003397
			10.0	0.989778	0.002269	0.008573	0.001718
		EWD	2.0	0.980873	0.002426	0.014440	0.001744
			3.0	0.981004	0.002468	0.014400	0.001761
			10.0	0.983665	0.004615	0.013214	0.003023
		IQR	0.0	0.985111	0.003816	0.011485	0.002218
		Median	0.0	0.986948	0.003063	0.009587	0.002153
		SRAD	0.0	0.990574	0.002766	0.007499	0.002001
	Humidity	EFD	2.0	0.984339	0.004027	0.011745	0.002634
			3.0	0.988943	0.004098	0.008855	0.002733
			10.0	0.989633	0.001976	0.008137	0.001391
		EWD	2.0	0.985212	0.004234	0.011147	0.002674
			3.0	0.990516	0.001869	0.007969	0.001532
			10.0	0.989883	0.001772	0.008105	0.001406
		IQR	0.0	0.987393	0.003538	0.010017	0.002461
		Median	0.0	0.984339	0.004027	0.011745	0.002634
		SRAD	0.0	0.990516	0.001869	0.007966	0.001532
	Light	EFD	2.0	0.984325	0.002994	0.011917	0.002295
			3.0	0.985000	0.002973	0.010448	0.001880
			10.0	0.990634	0.002807	0.007148	0.002018
		IQR	0.0	0.990396	0.002649	0.007441	0.002104
		Median	0.0	0.984958	0.003274	0.010935	0.002341
		SRAD	0.0	0.990644	0.002789	0.007200	0.002315

Table E.6: Full scores of the room occupancy dataset ($n = 10$)

E.4 Synthetic Dataset

				Accuracy		Brier-Score	
				mean	std	mean	std
Dataset	Variable	Method	Bins				
Test	v	EFD	2.0	0.928829	0.008929	0.057969	0.005556
			3.0	0.898348	0.007592	0.069993	0.002570
			10.0	0.947798	0.005369	0.034903	0.002886
		EWD	2.0	0.905455	0.008950	0.068953	0.004129
			3.0	0.895045	0.007857	0.072270	0.002871
			10.0	0.919019	0.008183	0.044861	0.002817
		IQR	0.0	0.879479	0.009426	0.090399	0.002909
		Median	0.0	0.928829	0.008929	0.057969	0.005556
		SRAD	0.0	0.947798	0.005369	0.034903	0.002886
	w	EFD	2.0	0.939089	0.006514	0.038726	0.002898
			3.0	0.901952	0.007409	0.068241	0.002705
			10.0	0.960911	0.005068	0.023827	0.002099
		EWD	2.0	0.962763	0.005431	0.027593	0.003016
			3.0	0.892392	0.008651	0.074470	0.003516
			10.0	0.962663	0.005551	0.026810	0.003073
		IQR	0.0	0.873373	0.009864	0.089678	0.004568
		Median	0.0	0.939089	0.006514	0.038726	0.002898
		SRAD	0.0	0.961011	0.005150	0.023901	0.002101
Train	v	EFD	2.0	0.928838	0.000398	0.057965	0.000244
			3.0	0.898236	0.000434	0.069958	0.000151
			10.0	0.948006	0.000375	0.034662	0.000141
		EWD	2.0	0.905717	0.003366	0.068798	0.001520
			3.0	0.894953	0.000409	0.072251	0.000150
			10.0	0.918663	0.000523	0.044804	0.000445
		IQR	0.0	0.879398	0.000487	0.090342	0.000228
		Median	0.0	0.928838	0.000398	0.057965	0.000244
		SRAD	0.0	0.948006	0.000375	0.034662	0.000141
	w	EFD	2.0	0.939092	0.000447	0.038719	0.000135
			3.0	0.901854	0.000430	0.068256	0.000165
			10.0	0.960896	0.000446	0.023813	0.000096
		EWD	2.0	0.962686	0.002360	0.027588	0.001376
			3.0	0.892245	0.000455	0.074509	0.000212
			10.0	0.962689	0.002360	0.026485	0.001205
		IQR	0.0	0.873415	0.000483	0.089620	0.000198
		Median	0.0	0.939092	0.000447	0.038719	0.000135
		SRAD	0.0	0.961504	0.002650	0.023688	0.000542

Table E.7: Full results of the synthetic dataset ($n = 10$)

Dataset	Variable	Score	Score SRAD	Std. SRAD	Score Median	Std. Median	p
Train	v	Accuracy	0.948	0.000	0.929	0.000	0.0
Train	v	Brier-Score	0.035	0.000	0.058	0.000	0.0
Train	w	Accuracy	0.962	0.003	0.939	0.000	0.0
Train	w	Brier-Score	0.024	0.001	0.039	0.000	0.0
Test	v	Accuracy	0.948	0.005	0.929	0.009	0.0
Test	v	Brier-Score	0.035	0.003	0.058	0.006	0.0
Test	w	Accuracy	0.961	0.005	0.939	0.007	0.0
Test	w	Brier-Score	0.024	0.002	0.039	0.003	0.0

Table E.8: Comparison of results between S-RAD and Median, based on different scores and measured variables for the synthetic dataset.