Avoiding Overfitting in Additive Bayesian Networks

Master Thesis in Mathematics (MAT491)

by

Nicolas Huber
16-740-094

supervised by

Prof. Dr. Reinhard Furrer

Zurich, October 2021

Avoiding Overfitting in Additive Bayesian Networks

Nicolas Huber

Version November 12, 2021

Contents

Preface iii

Abstract v

1 Introduction 1
- 1.1 Linear Models . . . 1
- 1.2 Generalized Linear Models . . . 1
- 1.3 Additive Bayesian Networks . . . 2
- 1.4 Structure Learning . . . 3
- 1.5 Overfitting . . . 10

2 Bootstrapping 13

3 Trimming 27
- 3.1 Trimming G . . . 27
- 3.2 Comparison between Trimming Algorithms . . . 34
- 3.3 Trimming Initial Guess . . . 35
- 3.4 Simulation Studies . . . 36

4 Prediction based 41
- 4.1 Prediction Algorithm . . . 41
- 4.2 Estimating Markov Blanket . . . 45
- 4.3 Example . . . 50
- 4.4 Alternative Approach . . . 52
- 4.5 Example . . . 54
- 4.6 Simulation Study . . . 56

5 Discussion and Outlook 61

A R Code 65

Bibliography 85

Preface

I would like to thank my supervisor Prof. Dr. Reinhard Furrer for giving me the opportunity to write my master thesis under his supervision. I have benefited tremendously from our meetings. In addition I would like to thank my fellow students Stefan Willi, Philipp Moor, Elena Silingardi, Jonas Guler, David Berger, Stefan Kurz, Roger Bader, Carl Zürcher and Sarina Sutter.

Last but not least I would like to thank my family and Mara Milanesi for the never ending belief in my forte that helped me improve from day to day.

Nicolas Huber
November 2021

Abstract

Additive Bayesian networks are graphical models describing the dependency structure between random variables. We often fit models containing too many links between random variables which are not dependent on each other. So it would be helpful to have some indication which edges in a directed acylic graph (DAG) are more likely to describe the dependency structure of the probability distribution from which the data has been sampled. Therefore a single fit is not enough. We are interested in a collection of DAGs, which we can use to build confidence bands for each edge in the DAG. For this reason, Friedman *et al.* (2013) introduced a parametric bootstrapping algorithm for Additive Bayesian networks. After applying the parametric bootstrapping algorithm, one is left with a collection of DAGs that must be summarized into one single DAG. This can be done by counting the number of occurrences of each edge in the bootstrap samples. The more often an edge is apparent in the bootstrap samples, the more likely, the distribution from which the data has been sampled contains this particular edge.
An alternative approach to make use of the bootstrap samples is to estimate the Markov blankets of each vertex in the DAG by introducing a prediction algorithm for binary Additive Bayesian networks. For each bootstrap sample and each vertex, the prediction accuracy can be estimated. This way it is possible to estimate each Markov blanket.
All functions and simulations are implemented in the programming language R. Moreover, the packages `abn` (Kratzer *et al.*, 2019a), `coda` (Plummer *et al.*, 2006), `rjags` (Plummer, 2018) and `igraph` (Csardi and Nepusz, 2006) have extensively been used.

Chapter 1

Introduction

This chapter is mainly based on Pittavino (2016).

1.1 Linear Models

A classical linear regression model with p predictors, is given by

$$Y_i = \beta_0 + \beta_1 x_{i1} + \dots \beta_p x_i + \varepsilon_i = \boldsymbol{x}_i{}^\top \boldsymbol{\beta} + \varepsilon_i, \quad i = 1, \dots, n,$$

where

1. Y_i is the dependent variable,
2. $\boldsymbol{x}_i = (1, x_{i1}, \dots, x_{ip})^\top$ are the predictors,
3. $\boldsymbol{\beta} = (\beta_0, \dots, \beta_p)^\top$ are the parameters

such that

1. $Y_i \sim \mathcal{N}(\boldsymbol{x}_i{}^\top \boldsymbol{\beta}, \sigma^2)$,
2. $\text{Var}(\varepsilon_i) = \sigma^2$ for $i \leq n$,
3. $\text{Cov}(\varepsilon_j, \varepsilon_k) = 0$ for $j \neq k$.

1.2 Generalized Linear Models

As seen in Section 1.1, linear models are very restrictive. First of all, we assume $\boldsymbol{Y}$ to be Gaussian. Second of all, we assume $\mathbb{E}[\boldsymbol{Y}] = \boldsymbol{X}^\top \boldsymbol{\beta}$. The goal of generalized linear models is to relax both of these assumptions. In generalized linear models, we assume that the distribution of $\boldsymbol{Y}$ is no longer necessarily Gaussian but is a member of the exponential family. A distribution belongs to the exponential family if the probability density function can be written in the following form:

$$f_Y(y; \theta) = \exp\left(\frac{\theta y}{\phi} + c(y, \phi)\right)$$

such that θ is the parameter of interest and $b(\cdot)$ and $c(\cdot, \cdot)$ are some functions. Members of the exponential family are the Binomial, Poisson, Gaussian, Inverse-Gaussian, and Gamma distribution, among others. Additionally we no longer assume that $\mathbb{E}[\boldsymbol{Y}] = \boldsymbol{X}^\top \boldsymbol{\beta}$. We generalize it to $\mathbb{E}[\boldsymbol{Y}] = g^{-1}(\boldsymbol{X}^\top \boldsymbol{\beta})$ where g is the so-called the link function. Therefore, $\mathbb{E}[\boldsymbol{Y}]$ is no longer just a linear combination of the values of $\boldsymbol{X}$, but a transformed version of it.

1.3 Additive Bayesian Networks

Additive Bayesian networks (ABN) are probabilistic graphical models that describe dependency structures between random variables. The goal is to find a factorization of the joint probability distribution. If we are given a set of random variables $\boldsymbol{X} = \{X_1, ..., X_d\}$, an ABN consist of:

1. A directed acyclic graph (DAG) $\mathcal{S} = (V, E)$. A random variable X_j corresponds to vertex $j \in V = \{1, ..., d\}$. There is no distinction between the random variable X_j and vertex j. The set of edges E, denote the dependencies between random variables. If there is an edge going from vertex i to vertex j, then j depends on i.
2. A set of parents for a vertex j, denoted by $\mathbf{Pa}_j$.
3. A set of local probability distributions for all variables in the network called $\theta_{\mathcal{A}}$. Each vertex j, with parent set $\mathbf{Pa}_j$ is parametrized by local probability distribution $P(X_j \mid \mathbf{Pa}_j)$.

To make it more clear, let us make an example.

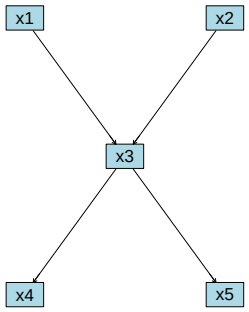


Figure 1.1: Plot of the DAG representing the dependency structure of some data.

The graph in Figure 1.1 does not contain any cycles, so it is a DAG. If we model the dependency structure using this DAG, then:

$$\mathbf{Pa}_1 = \emptyset,\ \mathbf{Pa}_2 = \emptyset,\ \mathbf{Pa}_3 = \{X_1, X_2\},\ \mathbf{Pa}_4 = \{X_3\},\ \mathbf{Pa}_5 = \{X_3\},$$

Moreover, the joint probability density function can be decomposed into the following product:

$$P(X_1, X_2, X_3, X_4, X_5) = P(X_1)P(X_2)P(X_3 \mid X_1, X_2)P(X_4 \mid X_3)P(X_5 \mid X_3),$$

where each factor is a generalized linear model.
The DAG in Figure 1.1 is just the structure. After learning the structure of the data, we also have to find the parameters $\theta_{\mathcal{A}}$ of our model. Since each factor of the joint probability distribution is a generalized linear model, the parameters of the above models are defined as follows:

- $\boldsymbol{\beta}^1 = \begin{pmatrix}\beta_{11} & 0 & 0 & 0 & 0\end{pmatrix}$ such that $\mathbb{E}[X_1] = g^{-1}(\beta_{11})$,
- $\boldsymbol{\beta}^2 = \begin{pmatrix}0 & \beta_{22} & 0 & 0 & 0\end{pmatrix}$ such that $\mathbb{E}[X_2] = g^{-1}(\beta_{22})$,
- $\boldsymbol{\beta}^3 = \begin{pmatrix}\beta_{31} & \beta_{32} & \beta_{33} & 0 & 0\end{pmatrix}$ such that $\mathbb{E}[X_3] = g^{-1}(\beta_{33} + X_1\beta_{31} + X_2\beta_{32})$,
- $\boldsymbol{\beta}^4 = \begin{pmatrix}0 & 0 & \beta_{43} & \beta_{44} & 0\end{pmatrix}$ such that $\mathbb{E}[X_4] = g^{-1}(\beta_{44} + X_3\beta_{43})$,
- $\boldsymbol{\beta}_5 = \begin{pmatrix}0 & 0 & \beta_{53} & 0 & \beta_{55}\end{pmatrix}$ such that $\mathbb{E}[X_5] = g^{-1}(\beta_{55} + X_3\beta_{53})$.

g are the link functions for the generalized linear models. $\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^5$ can be thought of vectors of dimension d where β^i_j is the parameter, corresponding to vertex i and parent j. Whenever

$\beta_j^i = 0$, then there does not exist an edge going from j to i. Moreover β_i^i is the intercept term of vertex i. This notation will be used for the remaining chapters. Therefore, an ABN is defined by $\mathcal{A} = (\mathcal{S}, \theta_{\mathcal{A}})$ where $\mathcal{S}$ is the structure and $\theta_{\mathcal{A}} = \{\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^5\}$ the parameters.

1.4 Structure Learning

Learning an ABN always consists of two parts. At first, one has to learn the structure of the data or, in other words, the DAG. The second part consists of learning the parameters of the network. In this paragraph, we are only covering the structure learning part. The general idea to learn a DAG $\mathcal{S}$ is to introduce some score function. This score function takes as an input a data set $\mathcal{D}$ and a DAG $\mathcal{S}$ and returns a score. The score of a DAG is a measure of how likely it is that the given data $\mathcal{D}$ has been sampled from the given DAG $\mathcal{S}$.

One popular choice is the marginal likelihood function (MacKay, 1992)

$$P(\mathcal{S} \mid \mathcal{D}) = \frac{P(\mathcal{S})P(\mathcal{D} \mid \mathcal{S})}{P(\mathcal{D})} \propto P(\mathcal{S})P(\mathcal{D} \mid \mathcal{S}),$$

where $P(\mathcal{S})$ is the prior distribution in the space of DAGs. Usually, this prior is chosen in an uninformative manner, meaning that we give every possible DAG the same weight. In other words, $P(\mathcal{S})$ is the uniform measure on the space of DAGs. In some cases, we might have some domain knowledge. For example, we might want to ban some of the edges, or we might want to force our network to have some edges in the graph in any case. Moreover, the maximal number of parent vertices per vertex could be restricted. In those cases, $P(\mathcal{S})$ gives the same probability to all DAGs satisfying our constraints.

The much harder part is the computation of $P(\mathcal{D} \mid \mathcal{S})$.

First notice that

$$P(\mathcal{D} \mid \mathcal{S}) = \int_{\boldsymbol{\beta}_{\mathcal{A}}} P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}_{\mathcal{A}})\pi(\boldsymbol{\beta}_{\mathcal{A}} \mid \mathcal{S})d\boldsymbol{\beta}_{\mathcal{A}}.$$

Meaning that the probability of getting the data that we have given our DAG equals the probability of getting the data given our structure and the parameters if the parameters are chosen according to the prior distribution of our parameters given our structure.

Notice that

$$P(\mathcal{D} \mid \mathcal{S}) = \prod_{i=1}^{d} P(\mathcal{D}_i \mid \mathcal{S})$$

where $\mathcal{D}_j$ is variable j.

Those probabilities are modeled via the link function. So for example if we are in the case of binary logistic ABNs, then

$$P(\mathcal{D}_j \mid \mathcal{S}) = \int_{\mathcal{B}} \prod_{i=1}^{n} \left(\frac{e^{\boldsymbol{x}_i{}^\top \boldsymbol{\beta}^j}}{1 + e^{\boldsymbol{x}_i{}^\top \boldsymbol{\beta}^j}} \right)^{x_{ij}} \left(\frac{1}{1 + e^{\boldsymbol{x}_i{}^\top \boldsymbol{\beta}^j}} \right)^{1-x_{ij}} \pi(\boldsymbol{\beta}^j \mid \mathcal{S})d\boldsymbol{\beta}^j \tag{1.1}$$

where $\boldsymbol{\beta}^j$ is the parameter vector corresponding to vertex j. This means that coordinate i of $\boldsymbol{\beta}^j$ is the parameter corresponding to the edge going from vertex i to vertex j. If i is not a parent of j then coordinate i of $\boldsymbol{\beta}^j$ is zero. Moreover, $\boldsymbol{x}_i$ is data point i and x_{ij} is the value of variable j of data point i. Lastly $\mathcal{B}$ is the parameter space in which $\boldsymbol{\beta}^j$ lives.

At this point, it is not obvious how the marginal likelihood penalizes too complex models. Usually, in those situations, a regularizer is added to penalize models containing too many edges. We have to dive into how the marginal likelihood is computed in the background to see where this penalization term comes up. Let us have a closer look at the computation of $P(\mathcal{D} \mid \mathcal{S})$.

$$P(\mathcal{D} \mid \mathcal{S}) = \int P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}) P(\boldsymbol{\beta} \mid \mathcal{S}) d\boldsymbol{\beta} \tag{1.2}$$

where

$$\boldsymbol{\beta} \mid \mathcal{S} \sim (2\pi)^{-k/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp\left(-\frac{1}{2} (\boldsymbol{\beta} - \boldsymbol{\beta}^*)^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{\beta} - \boldsymbol{\beta}^*) \right).$$

Notice that

$$P(\mathcal{D} \mid \boldsymbol{\beta}, \mathcal{S}) P(\boldsymbol{\beta} \mid \mathcal{S}) \propto \frac{P(\boldsymbol{\beta} \mid \mathcal{D}, \mathcal{S})}{P(\boldsymbol{\beta} \mid \mathcal{S})} P(\boldsymbol{\beta} \mid \mathcal{S}) = P(\boldsymbol{\beta} \mid \mathcal{D}, \mathcal{S}).$$

For large enough data sets $P(\boldsymbol{\beta} \mid \mathcal{D}, \mathcal{S})$, has a strong peak at the most probable parameter for the following reason (Held and Bové, 2020). Let us start with the discrete case.

Theorem 1.1. Let $\Theta = \{\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots\}$ be the parameter space with $\boldsymbol{\theta} \in \Theta$ be the unknown parameter such that $p_i = f(\boldsymbol{\theta}_i) = \mathrm{P}(\boldsymbol{\theta} = \boldsymbol{\theta}_i)$. Let $\boldsymbol{x}_{1:n}$ be the data sample and $\boldsymbol{\theta}_0$ be the true value of $\boldsymbol{\theta}$. Then

$$\lim_{n \to \infty} f(\boldsymbol{\theta}_j \mid \boldsymbol{x}_{1:n}) = \begin{cases} 1, & \text{for } j = 0, \\ 0, & \text{for } j \neq 0. \end{cases}$$

Definition 1.1. $D(\boldsymbol{\theta}_0 \mid\mid \boldsymbol{\theta}_i) = \int f(\boldsymbol{x} \mid \boldsymbol{\theta}_0) \log\left(\frac{f(\boldsymbol{x} \mid \boldsymbol{\theta}_0)}{f(\boldsymbol{x} \mid \boldsymbol{\theta}_i)} \right) \mathrm{d}\boldsymbol{x} = \mathbb{E}_{X \mid \boldsymbol{\theta}_0} \left[\log\left(\frac{f(X \mid \boldsymbol{\theta}_0)}{f(X \mid \boldsymbol{\theta}_i)} \right) \right]$, called the Kullback-Leibner discrepancy is a measure on how different two probability density functions are.

Proof.

$$f(\boldsymbol{\theta}_i \mid \boldsymbol{x}_{1:n}) = \frac{f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}_i) f(\boldsymbol{\theta}_i)}{f(\boldsymbol{x}_{1:n})} = \frac{p_i f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}_i)}{\sum_{j=1}^{\infty} f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}_j) f(\boldsymbol{\theta}_j)} = \frac{p_i f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}_i) / f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_0)}{\sum_{j=1}^{\infty} f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}_j) f(\boldsymbol{\theta}_j) / f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_0)}$$

$$= \frac{p_i \prod_{k=1}^{n} f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_i) / f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_0)}{\sum_{j=1}^{\infty} p_j \prod_{k=1}^{n} f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_j) / f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_i)} = \frac{\exp\left(\log(p_i) + S_i^{(n)} \right)}{\sum_{j=1}^{\infty} \exp\left(\log(p_i) + S_j^{(n)} \right)}$$

where $S_i^{(n)} = \sum_{k=1}^{n} \log\left(\frac{f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_j)}{f(\boldsymbol{x}_k \mid \boldsymbol{\theta}_0)} \right)$.

Notice that

$$D(\boldsymbol{\theta}_0 \mid\mid \boldsymbol{\theta}_0) = \int f(\boldsymbol{x} \mid \boldsymbol{\theta}_0) \log(1) = 0 \quad \text{and}$$

$$D(\boldsymbol{\theta}_0 \mid\mid \boldsymbol{\theta}_i) = \int f(\boldsymbol{x} \mid \boldsymbol{\theta}_0) \log\left(\frac{f(\boldsymbol{x} \mid \boldsymbol{\theta}_0)}{f(\boldsymbol{x} \mid \boldsymbol{\theta}_i)} \right) = \mathbb{E}_{X \mid \boldsymbol{\theta}_0} \left[\log\left(\frac{f(X \mid \boldsymbol{\theta}_0)}{f(X \mid \boldsymbol{\theta}_i)} \right) \right] > 0$$

since $i \neq 0$. At the same time

$$\lim_{n \to \infty} \frac{1}{n} S_j^{(n)} = \mathbb{E}_{X \mid \boldsymbol{\theta}_0} \left[\log\left(\frac{f(X \mid \boldsymbol{\theta}_0)}{f(X \mid \boldsymbol{\theta}_j)} \right) \right] = -D(\boldsymbol{\theta}_0 \mid\mid \boldsymbol{\theta}_j) = \begin{cases} = 0, & \text{for } j = 0, \\ < 0, & \text{for } j \neq 0. \end{cases}$$

Therefore,

$$\lim_{n\to\infty} S_j^{(n)} = \begin{cases} 0, & \text{for } j = 0, \\ -\infty, & \text{for } j \neq 0. \end{cases}$$

Hence,

$$\lim_{n\to\infty} f(\boldsymbol{\theta}_i \mid \boldsymbol{x}_{1:n}) = \begin{cases} 1, & \text{for } i = 0, \\ 0, & \text{for } i \neq 0. \end{cases}$$

□

In the discrete case, we cannot expect an asymptotically normal posterior distribution. But the mass gets more and more concentrated around the true parameter if the sample size n tends to infinity.

Theorem 1.2. Let $\boldsymbol{\theta}$ be an unknown continuous parameter with datasample $\boldsymbol{x}_{1:n}$. Suppose that $\boldsymbol{\theta}$ is Gaussian distributed. Then $\boldsymbol{\theta} \mid \boldsymbol{x}_{1:n}$ is approximately Gaussian.

Proof.

$$f(\boldsymbol{\theta} \mid \boldsymbol{x}_{1:n}) \propto f(\boldsymbol{\theta}) f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}) = \exp(\underbrace{\log(f(\boldsymbol{\theta}))}_{(1)} + \underbrace{\log(f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta}))}_{(2)}).$$

$$(1) = \log(f(\boldsymbol{\theta})) \approx \log(f(\boldsymbol{m}_0)) - \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*)^\top \boldsymbol{I}_0 (\boldsymbol{\theta} - \boldsymbol{\theta}^*)$$

such that

$$\boldsymbol{I}_0 = \mathcal{H}(-\log(f(\boldsymbol{\theta}))).$$

for $\mathcal{H}(f)$ being the Hessian matrix of f. Notice that in the Taylor-expansion the linear term is not apparent. This is because we center the Taylor-expansion around $\boldsymbol{\theta}^*$ which is a maximum, hence the gradient is 0.

$$(2) = \log(f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta})) \approx \log(f(\boldsymbol{x}_{1:n} \mid \widehat{\boldsymbol{\theta}}_n)) - \frac{1}{2}(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}_n)^\top \boldsymbol{I}(\widehat{\boldsymbol{\theta}}_n)(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}_n)$$

such that $\widehat{\boldsymbol{\theta}}_n$ is the maximum likelihood estimation of $\boldsymbol{\theta}$ and $\boldsymbol{I}(\widehat{\boldsymbol{\theta}}_n) = \mathcal{H}(-\log(f(\boldsymbol{x}_{1:n} \mid \boldsymbol{\theta})))$. Therefore,

$$\begin{aligned} f(\boldsymbol{\theta} \mid \boldsymbol{x}_{1:n}) &\approx \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*)^\top \boldsymbol{I}_0 (\boldsymbol{\theta} - \boldsymbol{\theta}^*) - \frac{1}{2}(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}_n)^\top \boldsymbol{I}(\widehat{\boldsymbol{\theta}}_n)(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}_n)\right) \\ &= \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{m}_n)^\top \boldsymbol{I}_n (\boldsymbol{\theta} - \boldsymbol{m}_n)\right), \end{aligned}$$

where

$$\boldsymbol{I}_n = \boldsymbol{I}_0 + \boldsymbol{I}(\widehat{\boldsymbol{\theta}}_n)$$

and

$$\boldsymbol{m}_n = \boldsymbol{I}_n^{-1}[\boldsymbol{I}_0 \boldsymbol{m}_0 + \boldsymbol{I}(\widehat{\boldsymbol{\theta}}_n)\widehat{\boldsymbol{\theta}}_n].$$

Therefore,

$$\boldsymbol{\theta} \mid \boldsymbol{x}_{1:n} \overset{\text{approx}}{\sim} \mathcal{N}(\boldsymbol{m}_n, \boldsymbol{I}_n^{-1}).$$

□

So, the integral (1.2) can be approximated via the following method:

$$P(\mathcal{D} \mid \mathcal{S}) = \int P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}) P(\boldsymbol{\beta}, \mathcal{S}) d\boldsymbol{\beta} \approx P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}^*) P(\boldsymbol{\beta}^* \mid \mathcal{S}) \Delta\boldsymbol{\beta} \tag{1.3}$$

The interpretation of this formula is that we approximate the area under the curve by a rectangle. The height h of the rectangle is $P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}^*) P(\boldsymbol{\beta}^* \mid \mathcal{S})$ where

$$\boldsymbol{\beta}^* = \arg\max_{\boldsymbol{\beta} \in \mathbb{R}} P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}^*) P(\boldsymbol{\beta}^* \mid \mathcal{S}).$$

The width of the rectangle is $\Delta\boldsymbol{\beta}$. Both of these quantities depend on the variance. If we have high variance, then the curve will be broader and shorter, hence a small h and a large $\Delta\boldsymbol{\beta}$. If, however, the variance is low, the curve will be narrow and tall in the middle. Therefore the height h is large while the width $\Delta\boldsymbol{\beta}$ is going to be small.

Let us suppose that $P(\beta \mid \mathcal{D}, \mathcal{S})$ to be Gaussian and β to be 1-dimensional. Let

$$f(\beta) = -\log\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\beta-\beta^*)^2}{2\sigma^2}}\right) = \frac{1}{2}\log(2\pi\sigma^2) + \frac{(\beta-\beta^*)^2}{2\sigma^2}.$$

Then

$$f'(\beta) = \frac{1}{\sigma^2}(\beta - \beta^*)$$

and

$$f''(\beta) = \frac{1}{\sigma^2}, \tag{1.4}$$

If we chose

$$\Delta\beta = \sqrt{\frac{2\pi}{f''(\beta)}} = \frac{\sqrt{2\pi}}{\sigma}.$$

it is possible to approximate the integral (1.3). We can generalize this even further for k-dimensional $\boldsymbol{\beta}$ vectors by choosing

$$\Delta\boldsymbol{\beta} = (2\pi)^{k/2} \det(\boldsymbol{A})^{-1/2}$$

where

$$\boldsymbol{A} = \mathcal{H}(-\log(P(\boldsymbol{\beta} \mid \mathcal{S}, \mathcal{D})))\big|_{\boldsymbol{\beta}=\boldsymbol{\beta}^*}.$$

Hence,

$$P(\mathcal{D} \mid \mathcal{S}) \approx P(\mathcal{D} \mid \boldsymbol{\beta}^*, \mathcal{S}) P(\boldsymbol{\beta}^* \mid \mathcal{S}) (2\pi)^{k/2} \det(\boldsymbol{A})^{-1/2}.$$

At the end we compute the logarithm of the marginal likelihood

$$\log(P(\mathcal{D} \mid \mathcal{S})) \approx \log(P(\mathcal{D} \mid \boldsymbol{\beta}^*, \mathcal{S})) + \log(P(\boldsymbol{\beta}^* \mid \mathcal{S})) + \frac{k}{2}\log(2\pi) - \frac{1}{2}\log(\det(\boldsymbol{A})).$$

The determinant of the Hessian matrix of $-\log(P(\boldsymbol{\beta} \mid \mathcal{S}, \mathcal{D}))$ works as a regularizer. In (1.4) we have seen that in the 1-dimensional case $f''(\beta)$ is directly linked to the variance of β. Now we want to investigate how in the multidimensional case, $\boldsymbol{A}$ is connected to the covariance matrix $\boldsymbol{\Sigma}$.

$$f(\boldsymbol{\beta}) = -\log(P(\boldsymbol{\beta} \mid \mathcal{D}, \mathcal{S})) = \frac{k}{2}\log(2\pi) + \frac{1}{2}\log(\det(\boldsymbol{\Sigma})) + \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}^*)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\beta} - \boldsymbol{\beta}^*).$$

Therefore,

$$\frac{\partial f}{\partial \boldsymbol{\beta}} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\beta} - \boldsymbol{\beta}^*).$$

Hence,

$$\frac{\partial^2 f}{\partial^2 \boldsymbol{\beta}} = \boldsymbol{\Sigma}^{-1}.$$

So the Hessian matrix relates the same way to the covariance of f as the second derivative relates to the variance. But why is the determinant of the inverse of the covariance matrix of $\boldsymbol{\beta}$ a good regularizer? Suppose $\boldsymbol{\Sigma} = \boldsymbol{I}_d$. This means no correlation at all for $\boldsymbol{\beta}$ which is equivalent to no correlation at all in the data itself. This is the equivalent of an empty graph containing no edges at all. Therefore, the determinant is 1 and the logarithm of the determinant is zero, which is equivalent to 0 penalization for the empty model. If there is a lot of correlation such that $\boldsymbol{\Sigma} = (1-\varepsilon)\mathbf{1}\mathbf{1}^\top + \varepsilon \boldsymbol{I}_d$ then, $\det(\mathcal{H}(A)) = (\varepsilon^{n-1} n(1-\varepsilon))^{-1}$. If ε tends to 0, which is equivalent to very strong correlation, the penalty term explodes to infinity.

Other very popular choices for score functions are the BIC and AIC, defined as

$$\begin{aligned}\text{BIC} &= k\ln(n) - \ln(\widehat{L}),\\ \text{AIC} &= 2k - \ln(\widehat{L}),\end{aligned}$$

where $\widehat{L}$ is the maximized value of the likelihood function of the model $\mathcal{S}$, k is the number of edges in the network, and n being the number of data points in our data set. Both the AIC and BIC are being minimized to get the optimal model. Notice that the BIC punishes models with a large number of edges stronger than the AIC.

Now we are finally ready to introduce the fitting algorithms.

Let us first define a function f that takes a DAG $\mathcal{S}$ and an operation E as an input and returns a DAG $f(\mathcal{S}, E)$. E, called the operation, can be one of three different things:

1. Edge addition: Inserts a single edge between two non-adjacent vertices such that the resulting graph is still a DAG.
2. Edge deletion: Removes one single edge between two vertices.
3. Edge reversal: Reverses the direction of one single edge such that the resulting graph is still a DAG.

A widespread way how models are fitted is to use a greedy algorithm (see R-Code 1.1). This means that the algorithm starts with some initial DAG, mostly the empty DAG.
Then it applies every possible operation $f(\mathcal{S}_0, E)$ to the empty DAG and checks in which case the score is highest.

$$\begin{aligned}\mathcal{S}_1 &= \arg\max_E \text{score}(f(\mathcal{S}_0, E)) \quad \text{and}\\ \mathcal{S}_{n+1} &= \begin{cases} f(\mathcal{S}_n, E_*), & \text{if score}(f(\mathcal{S}_n, E_*), \mathcal{D}) - \text{score}(\mathcal{S}_n, \mathcal{D}) > 0,\\ \mathcal{S}_n, & \text{else,}\end{cases}\end{aligned}$$

where $E_* = \arg\max_E \text{score}(f(\mathcal{S}_n, E))$
This greedy algorithm converges to some local minimum, but unfortunately, not necessarily to the global minimum. Let us make an example to illustrate why this algorithm does not necessarily converges to the global minimum.
Let us assume we have the following relationship between random variables.

$$X_1 \sim \text{Ber}(0.5)$$

$$X_2 \sim \text{Ber}(0.5)$$

$$X_3 = \begin{cases} 1, & \text{if } (X_1, X_2) = (0,1) \text{ or } (X_1, X_2) = (1,0), \\ 0, & \text{else.} \end{cases}$$

$$X_4 = \begin{cases} X_1, & \text{with probability } 0.6, \\ \text{Ber}(0.5), & \text{with probability } 0.4. \end{cases}$$

In the above example, one can easily see that the variables X_1 and X_2 on their own do not contain any information about X_3. But the two of them combined fully determine X_3. Therefore, when starting with the empty DAG, the algorithm will first add an edge between variable X_1 and X_4 since X_4 depends on X_1. In the second iteration step, likely, the algorithm will not add any new edges to the model so that our final DAG only consists of an edge between X_1 and X_4. The reason for that is that adding an edge from X_1 to X_3 does not improve the score of the network significantly, since the information of random variable X_2 is missing. The same logic can be applied to argue that the network score does not significantly improve when adding X_2 to the model. Only when adding both of them, the score is improving.

In R an ABN using a Greedy algorithm can be fitted as follows.
First, we need to load in some data and specify the distributions of our data set.

R-Code 1.1. Fitting ABN using greedy algorithm with marginal likelihood as score function.

```
data("pigs.vienna")
pigs.vienna <- pigs.vienna[,-11]

mydists <- list(PC = "binomial", PT = "binomial", MS = "binomial",
                HS = "binomial", TAIL = "binomial",
                Abscess = "binomial", Pyaemia = "binomial",
                EPcat = "binomial", PDcat = "binomial",
                plbinary = "binomial")

mycache <- buildScoreCache(data.df = pigs.vienna, data.dists = mydists,
                           method = 'bayes', max.parents = 6)

dag <- searchHillClimber(score.cache = mycache, score = 'mlik',
                         start.dag = NULL)
```

An easy way to not land in a local minimum, but in the global is by conducting a complete search, meaning we consider every possible DAG and return the best scoring one. The problem with this approach is that it is computationally too demanding since the number of possible DAGs grows faster than exponential with the number of vertices. So this only works for data sets with a small number of variables.
In R, an exact search with marginal likelihood as the score function can be conducted using R-Code 1.2.

R-Code 1.2. Fitting ABN using exact search with marginal likelihood as score function.

```
mycache <- buildScoreCache(data.df = pigs.vienna, data.dists = mydists,
                           method = 'bayes', max.parents = 6)

dag <- mostProbable(score.cache = mycache, score = 'mlik',
                    start.dag = NULL)
```

An exact search using the BIC score function can be conducted using R-Code 1.3.

R-Code 1.3. Fitting ABN using exact search with BIC as score function.

```
mycache <- buildScoreCache(data.df = pigs.vienna, data.dists = mydists,
                           method = 'mle', max.parents = 6)

dag <- mostProbable(score.cache = mycache, score = 'bic',
                    start.dag = NULL)
```

The third approach is quite similar to the greedy search but with one crucial difference. Instead of starting with the empty DAG, we start with some initial non-empty DAG and run it multiple times with different initial DAGs. After some iterations, we pick the DAG with the highest score. The advantage of this approach is that it might find a local minimum that would never be found by the greedy algorithm starting with the empty DAG. On the other hand, this algorithm might need many iterations until it finds a local minimum with a better score than the one obtained by the greedy search.

In R, this can be established as shown in R-Code 1.4.

R-Code 1.4. Fitting ABN using greedy algorithm with multiple restarts.

```
mycache <- buildScoreCache(data.df = pigs.vienna, data.dists = mydists,
                           method = 'bayes', max.parents = 6)

dag <- searchHillClimber(score.cache = mycache, score = 'mlik',
                         start.dag = NULL, num.searches = 1000)
```

1.5 Overfitting

A huge issue when fitting ABNs is overfitting. As an illustrative example, let us consider

$$\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{10} \sim \mathcal{N}_5(\boldsymbol{0}, \boldsymbol{I}_5).$$

Let us estimate the covariance matrix $\boldsymbol{\Sigma}$ using $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{10}$.

R-Code 1.5. Illustration of overfitting.

```
library(MASS)

set.seed(2)
n <- 25
mu <- rep(0, 5)
Sigma <- diag(5)
X <- mvrnorm(n, mu, Sigma)

cov(X)
```

```
##             [,1]        [,2]        [,3]        [,4]        [,5]
## [1,]  0.91940453  0.07933462 -0.23972516  0.09818682 -0.03357020
## [2,]  0.07933462  1.57497928 -0.30522010 -0.19184223 -0.42459771
## [3,] -0.23972516 -0.30522010  1.32421232  0.07583965  0.05073375
## [4,]  0.09818682 -0.19184223  0.07583965  1.15004900  0.23203637
## [5,] -0.03357020 -0.42459771  0.05073375  0.23203637  1.30936897
```

This example shows that the estimated covariance matrix indicates high covariance between variable 2 and 5, although it is a sample from a normal distribution with 0 covariance. Note that with $n = 25$ a correlation of ± 0.42 is barely significant at level $\alpha = 0.05$.This issue is due to the small sample size of our data set. We could redo the experiment for larger values of n, and the covariance matrix would converge to $\boldsymbol{I}_5$. ABNs suffer from the same issue. When dealing with small sample sizes, it often happens that the model overfits, meaning that dependencies between random variables that do not exist are believed to exist according to our ABN. So first, we want to define some measure on how much an ABN overfits the data.

Definition 1.2. Let $(\mathcal{S}, \theta_{\mathcal{A}})$ be an ABN such that $\mathcal{S} = (V, E)$ where $V = \{X_1, \ldots, X_d\}$. Then the Markov blanket of $Y \in V$ is defined as the parents of Y, the children of Y and the children parents of Y excluding Y itself. We denote the Markov blanket of a DAG $\mathcal{S}$ of vertex Y as $mb(Y, \mathcal{S})$.

Notice that $Y \perp\!\!\!\perp V \setminus mb(V, \mathcal{S}) \mid mb(V, \mathcal{S})$, which in words means, that given the Markov blanket, the value of Y is independent of $V \setminus mb(V, \mathcal{S})$.

One way, to measure how much a fitted model $\mathcal{S}^0$ overfits, given the true distribution $\mathcal{S}^t$, uses Markov blankets. For each vertex $X \in V$, one could compute the Markov blankets $mb(X, \mathcal{S}^0)$ and $mb(X, \mathcal{S}^t)$. Then we can construct a confusion matrix $\boldsymbol{M}$ such that

- M_{11} is the number of vertices, that are in $mb(X, \mathcal{S}^0)$ and $mb(X, \mathcal{S}^t)$ or in other words the true positives.
- M_{12} is the number of vertices that have incorrectly been classified as a member of the Markov blanket, or in other words, the false positives.
- M_{21} is the number of vertices that should have been in a certain Markov blanket but have not been found, or in other words the false negatives.

- M_{22} is the number of vertices that, in truth, are not in the Markov blanket and have not been classified as members of the Markov blanket.

Notice that the smaller M_{12} is, the less likely it is that our model overfits since it rarely finds an edge, that in truth, should not be there. At the same time, the larger M_{21}, the more likely, that the model overfits since it finds plenty of relations that are not there.

Chapter 2

Bootstrapping

A well-established method to prevent overfitting in ABNs is to use parametric bootstrapping. This idea was originally introduced by Friedman *et al.* (2013) and has further been discussed in Kratzer *et al.* (2019b).
The basic idea is to use an initial model $\mathcal{S}^0$ and sample N different data sets of the same size as the original observed data from it. In a second step, we refit N new models $\mathcal{S}^1, \ldots \mathcal{S}^N$ using the previously generated data sets. Let us have a closer look at how the bootstrapping works in more detail with an implementation in R. We used functionalities from the R packages *abn* (Kratzer *et al.*, 2019a), *coda* (Plummer *et al.*, 2006) and *rjags* (Plummer, 2018).

First, we load in some data, specify the distributions and make an initial guess *mp.dag* about the structure, using the function *mostProbable()*.

R-Code 2.1. Fitting ABN using exact search.

```
data("pigs.vienna")
pigs.vienna <- pigs.vienna[,-11]

mydists <- list(PC = "binomial", PT = "binomial", MS = "binomial",
                HS = "binomial", TAIL = "binomial",
                Abscess = "binomial", Pyaemia = "binomial",
                EPcat = "binomial", PDcat = "binomial",
                plbinary = "binomial")

mycache <- buildScoreCache(data.df = pigs.vienna, data.dists = mydists,
                           method = 'bayes', max.parents = 6)

mp.dag <- mostProbable(score.cache = mycache)
```

To sample from *mp.dag*, we need to compute the marginal distributions of the parameters of our model. The prior distribution of the parameters $\boldsymbol{\beta}$ is typically assumed to either be Dirichlet as described by Pittavino (2016) or Gaussian. Let us have a closer look at both of those possibilities.

Definition 2.1. Let $\boldsymbol{\beta} = (\beta_1, \dots, \beta_K) \sim \text{Dir}(\boldsymbol{\alpha})$ where $\boldsymbol{\alpha} \in \mathbb{R}_+^K$ be a Dirichlet random variable. Then the probability density function of $\boldsymbol{\beta}$ can be expressed as

$$P(\boldsymbol{\beta}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{l=1}^{K} \beta_l^{\alpha_l - 1},$$

where

$$\alpha_1, \dots, \alpha_K, \beta_1, \dots \beta_K > 0$$

and

$$\sum_{l=1}^{K} \beta_l = 1.$$

To simplify the notation, let $\boldsymbol{\beta} = (\beta_1, \dots, \beta_K)$ be the parameter vector of interest. Moreover let $z_j(\boldsymbol{\beta}, x_i) = \boldsymbol{x}_i{}^\top \boldsymbol{\beta}^j$. Our goal is to compute the marginal posterior distribution $P(\beta_m \mid \mathcal{D}, \mathcal{S})$ with two different priors on $\boldsymbol{\beta} \mid \mathcal{S}$. In the first case, we assume

$$\boldsymbol{\beta} \mid \mathcal{S} \sim \frac{1}{B(\boldsymbol{\alpha})} \prod_{l=1}^{K} \beta_l^{\alpha_l - 1}.$$

In the second case we assume

$$\boldsymbol{\beta} \mid \mathcal{S} \sim (2\pi)^{-K/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp\left(-\frac{1}{2} (\boldsymbol{\beta} - \boldsymbol{\beta}^*)^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{\beta} - \boldsymbol{\beta}^*) \right)$$

Let us start with case 1.

$$P(\boldsymbol{\beta} \mid \mathcal{S}, \mathcal{D}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{l=1}^{K} \beta_l^{\alpha_l - 1}.$$

Moreover,

$$P(\mathcal{D} \mid \boldsymbol{\beta}, \mathcal{S}) = \prod_{j=1}^{d} P(\mathcal{D}_j \mid \boldsymbol{\beta}, \mathcal{S}).$$

Let us further assume to have binary data. Then,

$$P(\mathcal{D}_j \mid \mathcal{S}, \boldsymbol{\beta}) = \prod_{i=1}^{n} \left(\frac{e^{z_j(\boldsymbol{\beta}, x_i)}}{1 + e^{z_j(\boldsymbol{\beta}, x_i)}} \right)^{x_{ij}} \left(\frac{1}{1 + e^{z_j(\boldsymbol{\beta}, x_i)}} \right)^{1 - x_{ij}} \pi(\boldsymbol{\beta} \mid \mathcal{S}).$$

By Bayes rule

$$P(\boldsymbol{\beta} \mid \mathcal{S}, \mathcal{D}) = \frac{P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}) P(\boldsymbol{\beta} \mid \mathcal{S})}{P(\mathcal{S} \mid \mathcal{D})} \propto P(\mathcal{D} \mid \mathcal{S}, \boldsymbol{\beta}) P(\boldsymbol{\beta} \mid \mathcal{S}).$$

Summarizing all the above equations, we end up with

$$P(\boldsymbol{\beta} \mid \mathcal{S}, \mathcal{D}) \propto \frac{1}{B(\boldsymbol{\alpha})} \prod_{l=1}^{K} \beta_l^{\alpha_l - 1} \prod_{i=1}^{n} \prod_{j=1}^{d} \left(\frac{e^{z_j(\boldsymbol{\beta}, x_i)}}{1 + e^{z_j(\boldsymbol{\beta}, x_i)}} \right)^{x_{ij}} \left(\frac{1}{1 + e^{z_j(\boldsymbol{\beta}, x_i)}} \right)^{1 - x_{ij}} \pi(\boldsymbol{\beta} \mid \mathcal{S}).$$

In order to end up with an expression for $P(\beta_m \mid \mathcal{D}, \mathcal{S})$, we have to integrate over the simplex S:

$$P(\beta_m \mid \mathcal{S}, \mathcal{D}) \propto \frac{1}{B(\boldsymbol{\alpha})} \int_S \prod_{l=1}^{K} \beta_l^{\alpha_l - 1} \prod_{i=1}^{n} \prod_{j=1}^{d} \left(\frac{e^{z_j(\boldsymbol{\beta}, x_i)}}{1 + e^{z_j(\boldsymbol{\beta}, x_i)}} \right)^{x_{ij}} \left(\frac{1}{1 + e^{z_j(\boldsymbol{\beta}, x_i)}} \right)^{1 - x_{ij}} \pi(\boldsymbol{\beta} \mid \mathcal{S}) d\boldsymbol{\beta}_{-m}.$$

On the other hand, if we assume

$$\boldsymbol{\beta} \mid \mathcal{S} \sim (2\pi)^{-K/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\beta}^*)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\beta}-\boldsymbol{\beta}^*)\right)$$

then

$$P(\beta_m \mid \mathcal{S}, \mathcal{D}) \propto \int_S p(\boldsymbol{\beta}) \prod_{i=1}^{n}\prod_{j=1}^{d} \left(\frac{e^{z_j(\boldsymbol{\beta},x_i)}}{1+e^{z_j(\boldsymbol{\beta},x_i)}}\right)^{x_{ij}} \left(\frac{1}{1+e^{z_j(\boldsymbol{\beta},x_i)}}\right)^{1-x_{ij}} \pi(\boldsymbol{\beta} \mid \mathcal{S}) d\boldsymbol{\beta}_{-m}$$

for

$$p(\boldsymbol{\beta}) = (2\pi)^{-K/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\beta}^*)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\beta}-\boldsymbol{\beta}^*)\right).$$

In practice, we first discretize each posterior density across a fine grid. Then for each y in the grid, $P(\beta_i = y \mid \mathcal{S}, \mathcal{D})$ is evaluated as above. We end up with K different densities. The function *fitAbn()* called in R-Code 2.2, automatically computes the posterior marginal densities of all parameters, meaning that each density is evaluated at *n.grid=1000* points. The interval in which the grid points are evaluated does not need to be specified since an algorithm called by the function computes it automatically.

R-Code 2.2. Computing the posterior marginal densities.

```
fit <- fitAbn(mp.dag, compute.fixed = T, n.grid = 1000)
```

R-Code 2.3 extracts the marginal densities from the *fit* object. Due to numerical issues, some density values $p(x)$ might be negative, and this is fixed by just setting every negative value to zero. An R file of those marginal densities is constructed. If we run this file, it would construct d different matrices, such that column $2i+1$ of matrix l holds the grid values of the marginal posterior density of the ith non-zero coordinate of $\boldsymbol{\beta}^l$, using the notation introduced in (1.1) and column $2i+2$ contains the values of the posterior marginal density at the given grid points. The first two columns of the matrices contain the grid points and the posterior marginal density values of the intercept terms of variable l. Later in the process, we will be sampling values from those probability density functions.

R-Code 2.3. Construct R file of posterior marginal densities.

```
generate_post_params <- function(fabn) {
  marnew <- fabn$marginals[[1]]
  for (i in 2:length(fabn$marginals)) {
    marnew <- c(marnew, fabn$marginals[[i]])
  }
  return(marnew)
}

post_params_pos <- function(post_params) {
  for (i in 1:length(post_params)) {
    for (j in 1:(dim(post_params[[i]])[2])) {
      if (j%%2 == 0) {
        ind <- which(post_params[[i]][, j] < 0)
        post_params[[i]][ind, j] <- 0
```

```
      }
    }
  }
  return(post_params)
}

bind_params <- function(post_params) {
  # post_params: is generated by generate_post_params
  front <- c()
  back <- c()
  for (i in 1:length(post_params)) {
    front <- c(front, gsub("\\|.*", "", names(post_params)[i]))
    back <- c(back, gsub(".*\\|","",names(post_params)[i]))
  }

  output <- list()
  output <- sapply(unique(front), function(x) NULL)
  for (i in 1:length(post_params)) {
    output[[front[i]]] <- cbind(output[[front[i]]], post_params[[i]])
  }
  return(output)
}

construct_R_file <- function(fabn, path) {
  dists <- fabn$abnDag$data.dists
  post_params <- generate_post_params(fabn)
  post_params <- post_params_pos(post_params)
  params_binded <- bind_params(post_params)
  output.file <- sink(paste0(path, "/post_params.R"))
  col.names <- names(params_binded)
  for (i in 1:length(params_binded)) {
    if (dists[i] != 'gaussian') {
      num <- dim(params_binded[[i]])
      str <- paste0(rep(c('x', 'f(x)'), num[2]/2))
      cat(paste0(col.names[i], '.p <- structure(c(\n'), output.file)
      cat(paste0(params_binded[[i]], collapse = ',\n '), output.file)
      cat(paste0('), .Dim = c(', num[1], 'L', ', ', num[2], 'L', '),
          .Dimnames = list(NULL, c(', paste0(paste0('\"', str, '\"'),
                                             collapse = ', '), ')', ')',
          sep = ''), output.file)
      cat(')', output.file)
      cat('\n', output.file)
    } else if (dists[i] == 'gaussian') {
      num <- dim(params_binded[[i]])
      str <- paste0(rep(c('x', 'f(x)'), num[2]/2-1))
      cat(paste0(col.names[i], '.p <- structure(c(\n'), output.file)
      cat(paste0(params_binded[[i]][,-c(num[2]-1, num[2])],
                 collapse = ',\n '), output.file)
      cat(paste0('), .Dim = c(', num[1], 'L', ', ', num[2]-2, 'L', '),
```

```
            .Dimnames = list(NULL, c(', paste0(paste0('\"', str, '\"'),
                                               collapse = ', '), ')', ')',
            sep = ''), output.file)
        cat(')', output.file)
        cat('\n', output.file)

        str <- paste0(rep(c('x', 'f(x)'), 1))
        cat(paste0('prec.',col.names[i], '.p <- structure(c(\n'), output.file)
        cat(paste0(params_binded[[i]][,c(num[2]-1, num[2])], collapse = ',\n '),
            output.file)
        cat(paste0('), .Dim = c(', num[1], 'L', ', ', 2, 'L', '),
            .Dimnames = list(NULL, c(', paste0(paste0('\"', str, '\"'),
                                               collapse = ', '), ')', ')',
            sep = ''), output.file)
        cat(')', output.file)
        cat('\n', output.file)
      }
    }
    sink()
}
```

It is essential to first visually check the marginal densities estimated with the function *fitAbn()*. The function *plotDensities()* from R-Code 2.4 will take as an input the object obtained by applying *fitAbn()* and will plot the marginal posterior densities.

R-Code 2.4. Plot posterior marginal densities of the parameters obtained by applying *fitAbn()*.

```
plotDensities <- function(fabn) {
  post_params <- generate_post_params(fabn)
  post_params <- post_params_pos(post_params)
  for (i in 1:length(post_params)) {
    plot(post_params[[i]], type = 'l', main = names(post_params)[i])
  }
}
```

Notice that in Figure 2.1, all marginal posterior densities look Gaussian, which is a good sign. There might be cases where some of the densities do not look Gaussian, in which cases we have a data separation problem. In those cases, the bootstrap algorithm most likely will fail. If we run into such problems, it is essential to look at the data. Suppose we have binary data and the marginal posterior density of parameter β_{ij} does not look Gaussian. In that case, there is likely either no data point in the data set such that $X_i = 0$ and $X_j = 1$ or $X_i = 1$ and $X_j = 0$. In this case, one can add a data point such that for each possible configuration of X_i, X_j values, a data point exists.

Once we have the marginal distributions, the next step is to create the DAG in JAGS. This involves creating a BUG file, a file that contains a definition of our DAG in the language of JAGS. The DAG contained in the BUG file is the highest scoring structure. Creating the BUG file could, in principle, be created by hand. But we decided to automate this process as well. To

run this BUG file, later in the process, the above-created marginal densities are necessary. In each step, we sample parameter values from the estimated marginal distributions.

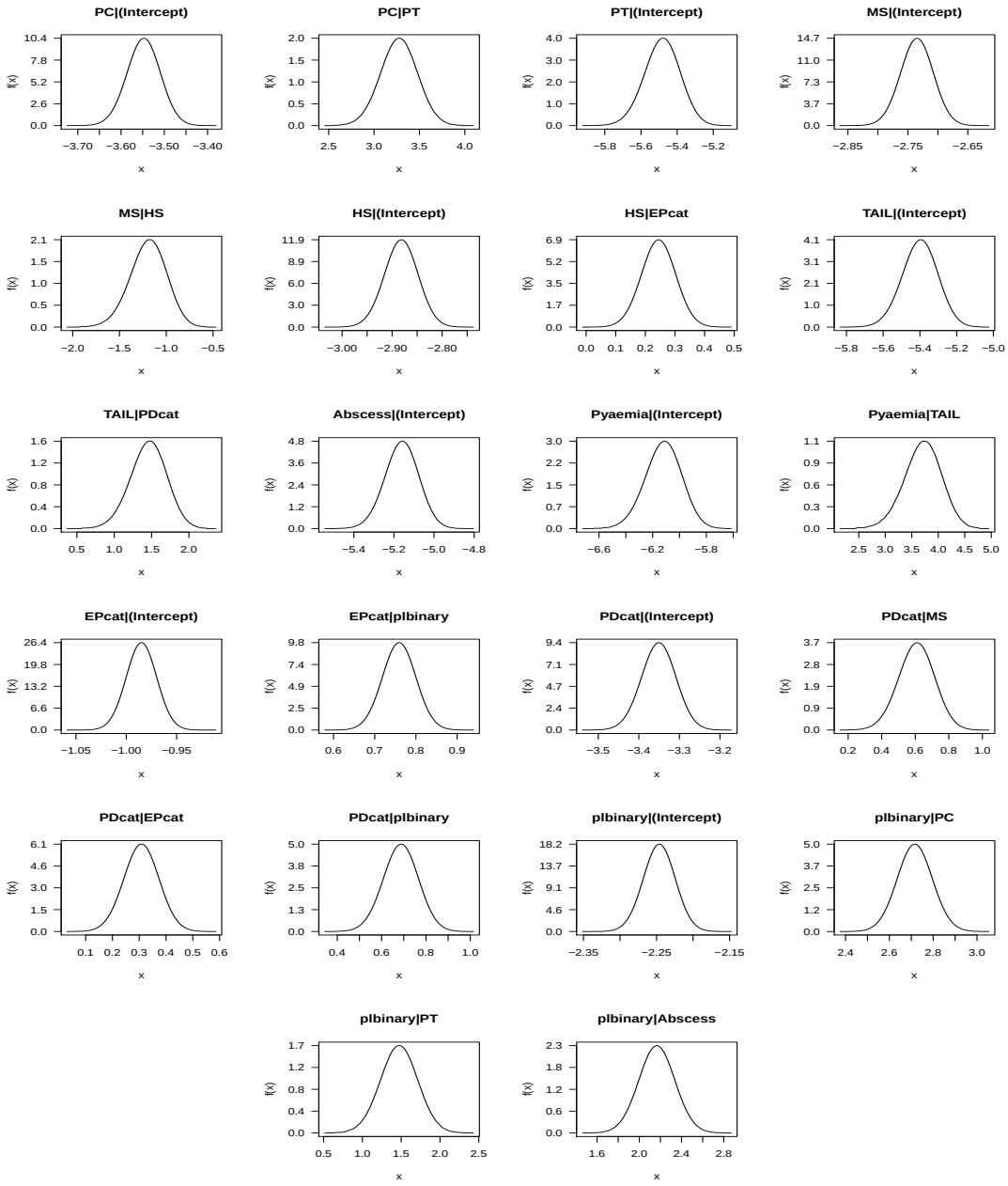


Figure 2.1: Plot of the posterior marginal distributions for the parameters of the ABN using R-Code 2.4.

R-Code 2.5. Create BUG file.

```
create_bug_file <- function(dag, dists, path) {
  # This function takes as an inputs:
  # - a dag (adjancency matrix)
  # - a named list of distributions
  # and returns:
  # - .bug file for MCMC simulations
```

```
# if (file.exists(paste0(path, "/model.bug"))) {
#   file.remove(path, "/model.bug")
# }
output.file <- sink(paste0(path, "/model.bug"))
cat("model{\n\n", output.file)
var.names <- names(dists)
for (i in 1:dim(dag)[1]) {
  form <- c(paste(var.names[i],'.b0', sep = ''))
  l <- length(var.names[i])
  dep <- c()
  dep <- c(dep, paste0(var.names[i], '.M0 ~ dcat(',
                       var.names[i],'.p[', '      ',2])'))
  dep <- c(dep, paste0(var.names[i], '.b0 <-    ',
                       var.names[i],'.p[',var.names[i],'.M0,1]'))

  cnt <- 1
  for (j in 1:dim(dag)[2]) {
    if (dag[i, j] != 0) {
      form <- c(form, paste0(' + ', var.names[i], '.b', cnt, '*',
                             var.names[j]))
      dep <- c(dep, paste0(var.names[i], '.M', cnt, ' ~ dcat(',
                           var.names[i], '.p[    ,', (cnt+1)*2,'])'))
      dep <- c(dep, paste0(var.names[i],'.b', cnt, ' <-     ',
                           var.names[i], '.p[',
                           var.names[i], '.M', cnt,',',2*(cnt+1)-1,']'))
      cnt <- cnt + 1
    }
  }
  if (dists[i] == 'gaussian') {
    dep <- c(dep, paste0('prec.',
                         var.names[i], '.M ~ dcat(prec.',
                         var.names[i],'.p[ ,2])', sep = ''))
    dep <- c(dep, paste0('prec.',
                         var.names[i], ' <- prec.',
                         var.names[i], '.p[prec.',var.names[i],'.M,1]'))
  }

  if (dists[i] == "binomial") {
    cat(paste0('\t', var.names[i], ' ~ dbern(p', i,');\n', sep = ''))
    cat(paste0('\tlogit(p', i, ') <- ', sep = ''))
  } else if (dists[i] == "poisson") {
    cat(paste0('\t', var.names[i], ' ~ dpois(p', i,');\n', sep = ''))
    cat(paste0('\tlog(p', i, ') <- ', sep = ''))
  } else if (dists[i] == 'gaussian') {
    cat(paste0('\t', var.names[i], ' ~ dnorm(mu.',
               var.names[i], ',prec.', var.names[i],');\n', sep = ''))
    cat(paste0('\tmu.', var.names[i], ' <- ', sep = ''))
  }
```

```
    for (k in 1:length(form)) {
      cat(paste0(form[k]))
    }
    cat(paste(';\n'))
    for (l in 1:length(dep)) {
      cat(paste0('\t', dep[l], ';'), output.file)
      cat(paste0('\n'))
    }
    cat(paste0('\n\n'))
  }
  cat(paste('\n\n}'))
  sink()
}
```

The user has the opportunity to return the DAGs, or the parameter matrix $\boldsymbol{\beta}$ such that β_{ij} is the parameter of the generalized linear model of vertex i, which is assigned to vertex j. The following two helper functions from R-Code 2.6 construct those parameter matrices.

R-Code 2.6. Parameter extractions.

```
construct_parameter_matrix <- function(params, data.params, d) {
  for (l in 1:d) {
    non.zero <- which(data.params[l,] == 1)
    for (m in seq(from = 1, to = dim(params[[l]])[2], by = 2)) {
      ind <- which.max(params[[l]][,m+1])
      if (m == 1) {
        data.params[l,l] <- params[[l]][ind, 1]
      } else {
        ind <- which.max(params[[l]][,m+1])
        data.params[l,non.zero[m]] <- params[[l]][ind, m]
      }
    }
  }
  return(data.params)
}

construct_parameter_matrix_single <- function(fabn, method = 'bayes', d) {
  mp.dag <- fabn$abnDag$dag
  data.params <- mp.dag + diag(d)
  for (l in 1:d) {
    if (method == 'bayes') tmp <- fabn$modes[[l]]
    else tmp <- tmp <- fabn$coef[[l]]
    non.zero <- which(data.params[l,] == 1)
    for (m in 1:length(tmp)) {
      if (m == 1) {
        data.params[l,l] <- tmp[m]
      } else {
        data.params[l, non.zero[m]] <- tmp[m]
      }
    }
```

```
  }
  return(data.params)
}
```

The last part is the bootstrapping part itself, which combines all the above-created functions. The bootstrapping function, from R-Code 2.7, starts with a data set. We use this data set to compute the highest-scoring DAG as a first initial guess $\mathcal{S}^0$ of the structure of the data. We then estimate the marginal densities and construct an R file out of them. Moreover, a BUG file is created, in which $\mathcal{S}^0$ is translated into the JAGS language. JAGS samples some data points $\mathcal{D}$, which are used to fit a new model. We repeat this process multiple times. This approach, however only works, when using the Bayesian approach. But we might want to do bootstrapping with a different score function, like the BIC or AIC. In those cases, we no longer sample parameters from the posterior marginal distributions, but we compute the maximum of this function and choose those parameters. We then sample data points using those particular parameters instead of sampling from the marginal posterior distribution. The remaining part of the bootstrapping algorithm works the same as if we would use the marginal likelihood as the score function.

R-Code 2.7. Bootstrapping algorithm.

```
bootstrap <- function(num.of.simulations,
                      data,
                      method = 'bayes',
                      score = 'mlik',
                      dists,
                      max.par,
                      ban = NULL,
                      retain = NULL,
                      n.chains,
                      n.iter,
                      n.adapt,
                      n.thin = 10,
                      mode = 'dag',
                      path = NULL,
                      abn.object = NULL,
                      bug.file = NULL,
                      post.params = NULL,
                      return.datasets = FALSE,
                      keep.files = FALSE) {

  # num.of.simulations =  Number of bootstrap samples
  # data =                The data
  # method =              With which method we want to fit abns.
  #                       Choices are: bayes or mle
  # score =               Score function: If method = bayes, then score = mlik
  #                                       If method = mle then score either
  #                                                  bic, aic, mlik or mdl
  # dists =               A named list of distributions of the variables
  # max.par =             The maximal number of parents (used for mycache)
  # ban =                 A matrix in which one encodes prior knowledge
```

```
#                         on which arcs are banned
# thin =                  The thin for the MCMC bootstrapping (default = 10)
# mode =                  'dag' returns dags
#                         'params' returns parameter matrices
# path =                  The path where all the files should be stored
# abn.object =            If there already exists an abn.object
#                         (including compute.fixed=TRUE and n.grid=n),
#                         then this can be used as an input since
#                         otherwise it will have to be computed
# bug.file =              Path to a bug file. If no path specified it will
#                         create automatically a bug file
# post.params =           Path to post.params.R file. If no path specified
#                         it will create automatically the R file
# return.datasets =       TRUE returns all datasets whenever a dag is fitted
# keep.files =            FALSE deletes all temporary files
#                         (only keeps post_params.R & model.bug)

n <- dim(data)[1]
d <- dim(data)[2]
dags <- list()

if (is.null(ban)) {
  ban <- matrix(rep(0, d^2), nrow = d)
}

if (is.null(retain)) {
  retain <- matrix(rep(0, d^2), nrow = d)
}

colnames(ban) <- rownames(ban) <- names(data)
colnames(retain) <- rownames(retain) <- names(data)

if (method == 'bayes') {
  if (score != 'mlik') {
    score <- 'mlik'
    warning('When using bayes to fit an abn, then only mlik is available
            as a score function. Score function has automatically been set
            to mlik.')
  }
}

if (is.null(path)) {
  path <- paste0('bootstrap_',
                 format(as.POSIXct(Sys.time(), tz = "UTC"),
                        "%F %R"), sep = '')
  path <- gsub(' ', '', path)
  path <- gsub(':', '', path)
  path <- gsub('-', '', path)
  dir.create(file.path(path), showWarnings = FALSE)
}
```

```
dir.create(file.path(path, '/tmp'), showWarnings = FALSE)
mycache <- buildScoreCache(data.df = data, data.dists = dists,
                           max.parents = max.par, method = method)
mp.dag <- mostProbable(score.cache = mycache, score = score)
marg.f <- fitAbn(object = mp.dag, data.df = data, data.dists = dists,
                 compute.fixed = TRUE, n.grid = 1000)

if (method == 'mle') {
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             max.parents = max.par, method = 'bayes',
                             dag.retained = mp.dag$dag,
                             dag.banned = 1-mp.dag$dag)
  mp.dag <- mostProbable(score.cache = mycache, score = 'mlik')
  marg.f <- fitAbn(object = mp.dag, data.df = data, data.dists = dists,
                   compute.fixed = TRUE, n.grid = 1000)
  post_params <- generate_post_params(marg.f)
  post_params <- post_params_pos(post_params)
  params <- bind_params(post_params)
  data.params <- mp.dag$dag

  data.params <- construct_parameter_matrix(params, data.params, d)
  if (mode == 'dag') {
    for (i in 1:num.of.simulations) {
      print(i)
      data.tmp <- simulateAbn(data.dists = dists, data.param = data.params,
                              n.chains = 1, n.adapt = n.adapt,
                              n.thin = n.thin, n.iter = n.iter,
                              verbose = FALSE, seed = i)
      mycache.tmp <- buildScoreCache(data.df = data.tmp, data.dists = dists,
                                     method = 'mle', dag.banned = ban,
                                     dag.retained = retain,
                                     max.parents = max.par, verbose = FALSE)
      mp.tmp <- mostProbable(score.cache = mycache.tmp,
                             score = score, verbose = FALSE)
      dags[[i]] <- mp.tmp$dag
    }
    return(dags)
  } else {
    for (i in 1:num.of.simulations) {
      print(i)
      data.tmp <- simulateAbn(data.dists = dists, data.param = data.params,
                              n.chains = 1, n.adapt = n.adapt,
                              n.thin = n.thin, n.iter = n.iter,
                              verbose = FALSE, seed = i)
      mycache.tmp <- buildScoreCache(data.df = data.tmp, data.dists = dists,
                                     method = 'mle', dag.banned = ban,
                                     dag.retained = retain,
                                     max.parents = max.par, verbose = FALSE)
      mp.tmp <- mostProbable(score.cache = mycache.tmp,
```

```
                                score = score, verbose = FALSE)
        fit.tmp <- fitAbn(object = mp.tmp, data.df = data.tmp,
                          data.dists = dists)
        params.tmp <- construct_parameter_matrix_single(fit.tmp, 'mle', d)
        dags[[i]] <- params.tmp
      }
      return(dags)
    }
  } else {
    if (is.null(bug.file))  {
      create_bug_file(marg.f$abnDag$dag, dists, path)
      bug.model.path <- paste0(path, '/model.bug')
    }
    else {
      bug.model.path <- bug.file
    }

    if (is.null(post.params)) {
      construct_R_file(marg.f, path)
      post.params.path <- paste0(path, '/post_params.R')
    }
    else {
      post.params.path <- post.params
    }

    if (method == 'mle') {
      post_params <- generate_post_params(marg.f)
      post_params <- post_params_pos(post_params)
      params <- bind_params(post_params)
    }

    start<-seq(1,2*num.of.simulations, by = 1)
    stop<-seq(num.of.simulations,2*num.of.simulations,by = 1)
    if (!exists('index')) index <- 1

    for(i in start[index]:stop[index]) {
      # create bootstrap data
      # 1. create parameter file with unique random seed
      init.file <- paste0(path, "/tmp/init_", i)  # file to hold jags seed
      cat(paste0("\".RNG.name\" <- \"base::Mersenne-Twister\"\n",
                 "\".RNG.seed\" <- ", i, "\n"), file = init.file)

      # 2. create script file with unique output file name
      run.file <- paste0(path, "/tmp/script_", i)   # file to hold jags seed
      out.file <- paste0(path, "/tmp/out_", i)

      monitor <- ''
      for (k in 1:d) {
        monitor <- paste0(monitor,
```

```
                        "monitor ", names(data)[k],
                        ", thin(", n.thin, ")\n", sep = "")
      }
      end <- paste0("update ", n.iter, ", by(1)
coda *, stem(\"", out.file, "\")\n", sep = "")

      ## this is needed verbatim (creates script_i file)
      cat(paste0("model in ", bug.model.path, "
data in ", post.params.path,
                 "\ncompile, nchains(", n.chains,")
parameters in ", init.file, "\n
initialize
update 100000\n", monitor, end), file = run.file, sep = "")

      # 3. run the MCMC sampler
      system(paste0("jags ", run.file))  # possibly adapt due to system

      # 4. read in mcmc data and convert to format suitable for mostprobable
      boot.data <- read.coda(paste0(out.file, "chain1.txt"),
                             paste0(out.file, "index.txt"))
      boot.data <- as.data.frame(boot.data)
      for(j in 1:dim(data)[2]){
        if(is.factor(data[,j])){
          boot.data[,j] <- as.factor(boot.data[,j])
          levels(data[,j]) <- levels(data[,j])
        }
      }

      # 5. run the MostProb search on the bootstrap data
      skip_to_next <- TRUE
      tryCatch(boot1.cache <- buildScoreCache(as.data.frame(boot.data),
                                              data.dists = dists,
                                              max.parents = max.par,
                                              dag.banned = ban,
                                              dag.retained = retain,
                                              method = method),
               error = function(e) {skip_to_next <<- TRUE})
      if (skip_to_next) {
        if (return.datasets == T & exists('boot1.cache')) {
          dags[[i]] <- list()
          graph <- mostProbable(score.cache = boot1.cache, score = score)$dag
          dags[[i]]$dag <- normalizeDag(graph)
          dags[[i]]$data <- boot.data
        } else if (return.datasets == T & !exists('boot1.cache')) {
          dags[[i]] <- NULL
        } else if (return.datasets == F & exists('boot1.cache')) {
          graph <- mostProbable(score.cache = boot1.cache, score = score)
          if (mode == 'dag') {
            dags[[i]] <- graph$dag
```

```
          } else {
            fit.tmp <- fitAbn(object = graph,
                              data.df = as.data.frame(boot.data),
                              data.dists = dists)
            params.tmp <- construct_parameter_matrix_single(fit.tmp)
            dags[[i]] <- params.tmp
          }
        } else if(return.datasets == F & !exists('boot1.cache')) {
          dags[[i]] <- NULL
        }
        else {
          dags[[i]] <- NULL
        }
        next
      }
      else {
        next
      }
    }
    save(dags,file = paste0(path, "/ResultBootstrap", index, ".RData"))
    if (!keep.files) {
      unlink(paste0(path, '/tmp', sep = ''), recursive = TRUE)
    }
    return(dags)
  }
}
```

Intuitively, our initial guess, according to the score function, is the DAG that best describes the dependency structures in our data. However, our bootstrap samples are DAGs generated from samples from this very high-scoring DAG. Therefore, all DAGs returned by the bootstrap algorithm are in a neighborhood of the global maximum of the landscape of DAGs. This collection might contain many models that describe the dependency structure of the underlying data even better than the initial DAG itself. The initial DAG might have some unstable edge due to some outlier in the original data set. But this edge would not be contained in many of the bootstrap samples.

One possibility to use this collection of DAGs is to use domain knowledge and take a look at the best scoring DAGs and only keep the ones that seem to make sense according to current research. But there might be cases when there are no experts available. Therefore, the subsequent chapter is dedicated to summarizing this collection of DAGs into one single DAG.

Chapter 3

Trimming

ABNs that overfit have too many edges. Therefore, we wish to start with some initial DAG $\mathcal{S}^0$, find all edges that represent dependencies that do not exist in the distribution from which the data has been sampled from and remove those. Hence, we need some criteria on when to drop an edge and when to keep it. This criterion is based on bootstrap samples. We start by some initial DAG $\mathcal{S}^0$. This initial DAG could be found via a greedy hillclimbing search or a complete search over the space of all DAGs. Let $\{G_k\}_{k=1}^N$ be the set of DAGs, the bootstrapping algorithm returns. Let

$$\mathcal{S}' = \frac{1}{N}\sum_{k=1}^{N} G_k.$$

$\mathcal{S}'_{ij}$ can be interpreted as the fraction of times an edge goes from j to i in our bootstrap samples. Let

$$G_{ij} = \begin{cases} 0, & \text{if } S_{ij} < t, \\ 1, & \text{else,} \end{cases}$$

for some threshold t such that $0 \leq t \leq 1$.

The resulting matrix G is an adjacency matrix for a graph. This graph comprises all edges which are apparent in proportion t of all DAGs $\{G_k\}_{k=1}^N$.
The larger we chose the threshold parameter t, the smaller the number of edges in G. So it makes sense to choose a larger threshold parameter if we know that the model strongly overfits or if the score function we choose does not control for overfitting. For example, when choosing AIC as a score function, we most likely have to choose a higher threshold than if we would select BIC as the score function since BIC punishes networks with higher edge density stronger than AIC. It is crucial to notice that G is not necessarily a DAG since it may contain cycles.

There are two ways how one can use G to avoid overfitting.

3.1 Trimming G

A trimming function takes as an input a graph G and returns a different graph $\mathcal{S}$ by removing edges from G. In this case, trimming is used to remove cycles, since in order for $(\mathcal{S}, \theta_{\mathcal{B}})$ to be a ABN, we need $\mathcal{S}$ to be acyclic. In the following, we present two algorithms to reach this goal.

Algorithm 3.1 (Greedy trimming algorithm)**.**

[0] Make a list L of all edges .

[1] For each $e \in L$ count the number of cycles which would be broken if we would remove e .

[2] Remove $e \in L$ which breaks maximal number of cycles.

[3] Repeat [0], [1] and [1] until all cycles are removed.

For this algorithm the library *igraph* (Csardi and Nepusz, 2006) has been used.

In R-Code 3.1, we define a function that makes a list of all cycles apparent in a graph.

R-Code 3.1. Returns all cycles in a graph.

```
library(igraph)
return_cycles <- function(graph) {
  # takes as an input some graph and returns a list of all cycles in the graph
  g <- graph_from_adjacency_matrix( graph )
  Cycles = NULL
  for(v1 in V(g)) {
    for(v2 in neighbors(g, v1, mode = "out")) {
      Cycles = c(Cycles,
                 lapply(all_simple_paths(g, v2,v1, mode = "out"),
                        function(p) c(v1,p)))
    }
  }
  Cycles <- Cycles[sapply(Cycles, min) == sapply(Cycles, `[`, 1)]
  return(Cycles)
}
```

The function defined in R-Code 3.2, computes the number of remaining cycles when removing a given edge. We repeat this procedure for each edge in the network and return the edge that removes the maximal number of cycles.

R-Code 3.2. Returns optimal edge to remove.

```
compute_ess_edges <- function(graph) {
  cycles <- return_cycles(graph)
  edges <- c()
  count <- c()
  n <- length(cycles)
  for (i in 1:n) {
    for (j in 1:(length(cycles[[i]])-1)) {
      ind <- find_pattern(cycles[[i]][j:(j+1)], edges)
      if (length(ind) > 0) {
        for (k in 1:length(ind)) {
          if (ind[k]%%2 == 1) {
            final.ind <- ind
            count[(ind+1)/2] <- count[(ind+1)/2] + 1
          }
        }
      }
```

```
      else {
        edges <- c(edges, cycles[[i]][j:(j+1)])
        count <- c(count, 1)
      }
    }
  }
  max.ind <- which(count == max(count))[1]
  return(edges[(2*max.ind-1):(2*max.ind)])
}
```

R-Code 3.3. Greedy trimming algorithm.

```
trim.greedy <- function(graph) {
  empty <- F
  cycles <- return_cycles(graph)
  if (length(cycles) == 0) {
    empty <- T
  }
  while(!empty) {
    edges <- compute_ess_edges(graph)
    graph[edges[1], edges[2]] <- 0
    cycles <- return_cycles(graph)
    if (length(cycles) == 0) {
      empty <- T
    }
  }
  return(graph)
}
```

This algorithm has the advantage that it is very fast. The disadvantage is that it is not data-driven. So, we might delete some edge that explains a lot of the dependency structure in the data just because removing it would break many cycles. The second algorithm is slower but takes the network scores into account. To understand this second algorithm, we first have to introduce a definition.

Definition 3.1. A bicycle is a cycle of length 2.

For each bicycle (i, j, i) either (i, j) or (j, i) needs to be removed. Since we have N bootstrap samples, we have to remove (i, j) if the associated scores satisfy $\mathcal{S}'_{ji} < \mathcal{S}'_{ij}$. Otherwise, we remove (j, i). Intuitively, this means that we remove the direction that has less evidence according to our bootstrap samples.

First, we create a list L of all edges that are apparent in any cycle. Then for every edge $e \in L$, we can either remove e, reverse e or keep e. Notice that by reversing e, we might construct new cycles. So, we have to account for that. Then for every edge e, we compute the average score of the network if we either keep, remove or reverse e. We choose the action with the highest average score. If the score is maximal if we reverse e, we only do so if the number of cycles decreases when reversing e. Otherwise, we choose the next best action. We repeat this process until there are no more cycles apparent.

R-Code 3.4. Score based trimming algorithm.

```
return_node_for_bicycle <- function(graph, node, bicycle, d) {
  # This functions returns for a given node all nodes that are connected to
  # the given node by a bicycle.
  node.bicycles <- c()
  for (j in 1:length(bicycle)) {
    r <- bicycle[j]%%d
    m <- (bicycle[j]-r)/d+1
    if (r == 0) {
      m <- m-1
      r <- d
    }
    if (r == node) {
      node.bicycles <- c(node.bicycles, m)
    } else if (m == node) {
        node.bicycles <- c(node.bicycles, r)
    }
  }
  return(node.bicycles)
}

subset_mlik <- function(dataset, comb, nodes) {
  if (is.null(comb) & is.null(nodes)) {
    return(dataset)
  }
  if (is.array(dataset)) {
    d <- dim(dataset)[2]
    tmp <- dataset
    if (length(comb) >= 2) {
      inter <- which(tmp[, nodes[1]] == comb[1])
      for (j in 2:length(comb)) {
        ind <- which(tmp[, nodes[j]] == comb[j])
        inter <- intersect(ind, inter)
      }
    } else {
      inter <- which(tmp[, nodes[1]] == comb[1])
    }
    dataset <- tmp[inter, ]
  }
  else {
    d <- length(dataset)
    for (i in 1:d) {
      tmp <- dataset[[i]]
      if (length(comb) >= 2) {
        inter <- which(tmp[, nodes[1]] == comb[1])
        for (j in 2:length(comb)) {
          ind <- which(tmp[, nodes[j]] == comb[j])
          inter <- intersect(ind, inter)
        }
```

```
      } else {
        inter <- which(tmp[, nodes[1]] == comb[1])
      }
      dataset[[i]] <- tmp[inter, ]
      }
    }
  return(dataset)
}

construct_mlik_list <- function(mycache, d) {
  out <- list()
  for (i in 1:d) {
    out[[i]] <- list()
    ind <- which(mycache$children == i)
    out[[i]] <- cbind(mycache$node.defn[ind, ], mycache$mlik[ind])
  }
  return(out)
}

trim.bicycle <- function(graph, mycache) {
  d <- dim(graph)[1]
  bicycle <- c()
  mlik_list <- construct_mlik_list(mycache, d)
  cyc <- T
  while (cyc) {
  # Here, we store the indices of the matrix that are bicycles
    cnt <- 0
    edges <- matrix(rep(0, d^2), nrow = d)
    for (i in 1:d) {
      for (j in i:d) {
        if (graph[i,j] == 1 & graph[j,i] == 1) {
          edges[i, j] <- edges[j, i] <- 1
          cnt <- cnt + 1
        }
      }
    }
    if (cnt == 0) {
      return(graph)
    }
    num.of.cyc <- rowSums(edges)
    mliks <- c()
    for (i in 1:d) {
      if (num.of.cyc[i] > 0) {
        m <- subset_mlik(mlik_list[[i]], NULL, NULL)
        mlik <- max(m[, d+1]) - min(m[, d+1])
        mliks <- c(mliks, mlik)
      }
      else {
        mliks <- c(mliks, 0)
```

```
      }
    }
    node <- which.max(mliks)
    num.of.bicycles <- sum(edges[,node])

    tmp <- rep(list(0:1), num.of.bicycles)
    comb <- expand.grid(tmp)
    neighbours <- which(edges[, node] > 0)
    mliks <- rep(0, dim(comb)[1])
    for (i in 1:dim(comb)[1]) {
      subset_comb <- as.matrix(subset_mlik(mlik_list[[node]],
                                           as.numeric(comb[i,]), neighbours),
                               ncol = d+1)
      if (length(subset_comb)>d+1) {
        mliks[i] <- mliks[i] + mean(subset_comb[, d+1])
      } else {
        mliks[i] <- mliks[i] + subset_comb[d+1]
      }
      for (j in 1:length(neighbours)) {
        subset_comb <- as.matrix(subset_mlik(
          mlik_list[[neighbours[j]]], (as.numeric(comb[i,j])+1)%%2, node),
                                 ncol = d+1)
        mliks[i] <- mliks[i] + mean(subset_comb[, d+1])
      }
    }
    ind <- which.max(mliks)
    perm <- comb[ind, ]
    for (i in 1:dim(comb)[2]) {
      if (perm[i] == 0) {
        graph[node, neighbours[i]] <- 0
      }
      else {
        graph[neighbours[i], node] <- 0
      }
    }
  }
  return(graph)
}


trim.general <- function(graph, mycache, d) {
  count <- length(return_cycles(graph))
  mlik_list <- construct_mlik_list(mycache, d)
  while (count != 0) {
    cycles <- return_cycles(graph)
    if (length(cycles) == 0) {
      return(graph)
    }
    # edges[,1] = from
```

```
    # edges[,2] = to
    # edges[,3] = num of cycles with (i,j)
    # edges[,4] = num of cycles with (j,i)
    # edges[,5] = mlik of (i,j)
    # edges[,6] = mlik of (j,i)
    # edges[,7] = mlik if neither (i,j) nor (j,i) in E
    edges <- matrix(ncol = 7)
    for (i in 1:length(cycles)) {
      for (j in 1:(length(cycles[[i]])-1)) {
        edge <- c(cycles[[i]][j+1], cycles[[i]][j])
        tmp.graph <- graph
        tmp.graph[edge[1], edge[2]] <- 1
        tmp.graph[edge[2], edge[1]] <- 0
        num.of.cyc <- length(return_cycles(tmp.graph))
        mliks.sta <- 0
        mliks.rev <- 0
        mliks.rem <- 0
        mlik.reverse <- subset_mlik(mlik_list[[edge[1]]], c(1), c(edge[2]))
        mlik.stay <- subset_mlik(mlik_list[[edge[2]]], c(1), c(edge[1]))
        mlik.remove <- subset_mlik(mlik_list[[edge[2]]], c(0), c(edge[1]))
        mliks.sta <- mean(mlik.stay[, d+1])
        mliks.rev <- mean(mlik.reverse[, d+1])
        mliks.rem <- mean(mlik.remove[, d+1])
        t <- min(na.omit(c(mliks.rev, mliks.sta, mliks.rem)))
        if (is.nan(mliks.rev)) {
          mliks.rev <- t
        }
        if (is.nan(mliks.sta)) {
          mliks.sta <- t
        }
        if (is.nan(mliks.rem)) {
          mliks.rem <- t
        }
        tmp <- c(cycles[[i]][j],
                 cycles[[i]][j+1],
                 length(cycles),
                 num.of.cyc,
                 mliks.sta,
                 mliks.rev,
                 mliks.rem)
        edges <- rbind(edges, tmp)
      }
    }
    # action = 1, reverse
    # action = 2, remove
    edges <- edges[-1, ]
    ind <- 0
    action <- 0
    mlik.tmp <- min(edges[1, c(5,6,7)])
```

```
    for (i in 1:(dim(edges)[1])) {
      if (((which.max(edges[i, c(6,7)])) == 1)
          & (edges[i, 3] > edges[i, 4])
          & (mlik.tmp < edges[i, 6])) {
        mlik.tmp <- edges[i, 6]
        action <- 1
        ind <- i
      } else if (((which.max(edges[i, c(6,7)])) == 2)
                 & (mlik.tmp < edges[i, 7])) {
        mlik.tmp <- edges[i, 7]
        action <- 2
        ind <- i
      }
    }
    if (action == 1) {
      graph[edges[ind, 1], edges[ind, 2]] <- 0
      graph[edges[ind, 2], edges[ind, 1]] <- 1
    }
    else if (action == 2) {
      graph[edges[ind, 1], edges[ind, 2]] <- 0
    }
  }
  return(graph)
}

trim.score.based <- function(graph, mycache) {
  d <- dim(graph)[1]
  graph <- trim.bicycle(graph, mycache)
  graph <- trim.general(graph, mycache, d)
  return(graph)
}
```

Notice that both algorithms given in R-Code 3.3 and R-Code 3.4 work very differently. One algorithm removes cycles so that we keep as many edges while removing all cycles. The other algorithm summarizes all DAGs to preserve the underlying structure in the data in the resulting DAG. To compare both trimming algorithms, we dedicate the next section to compare those trimming algorithms.

3.2 Comparison between Trimming Algorithms

To start we introduce the Hamming distance, a measure of distances between two DAGs.

Definition 3.2. The Hamming distance between $\mathcal{S}_1$ and $\mathcal{S}_2$ is the smallest number of operations that need to be applied to $\mathcal{S}_1$ to be equal to $\mathcal{S}_2$. An operation can be either

1. Edge addition: Inserts a single edge between two non-adjacent vertices such that the resulting graph is still a DAG.
2. Edge deletion: Removes a single edge between two vertices.
3. Edge reversal: Reverses the direction of a single edge such that the resulting graph is still a DAG.

We generate 240 DAGs $\{\mathcal{S}_i^j\}_{i=1,\dots,40}^{j=1,\dots,6}$, such that for all $i \leq 40$, $\mathcal{S}_k^i$ has edge density $k/10$. From each DAG in $\{\mathcal{S}_i^k\}_{i=1,\dots,40}$, we sample $300\,k^2$ data points. We use those data points to fit the DAGs $\{\widehat{\mathcal{S}}\}_{i=1,\dots,40}^{j=1,\dots,6}$. For each $i \in \{1,\dots,40\}$ and $j \in \{1,\dots,6\}$, we create a list L_{ij} which contains all edges that could be added to $\widehat{\mathcal{S}}_i^j$ such that the resulting graph contains at least one cycle. Then for each edge e in the list L_{ij}, we add e to $\widehat{\mathcal{S}}_i^j$ resulting the graph $\tilde{\mathcal{S}}_i^j$ and use both trimming algorithms, to remove the cycles. We then compute the Hamming distance between $\mathcal{S}_i^j$ and the Trimmed version of $\tilde{\mathcal{S}}_i^j$. Finally, we average the Hamming distances for each edge density.

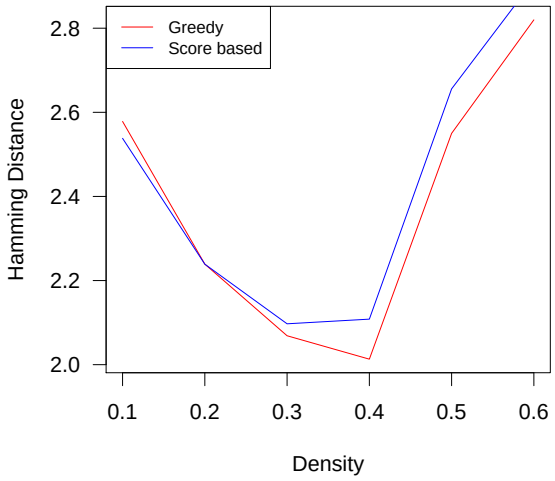


Figure 3.1: Plot of Hamming distance between DAG obtained by trimming down with the greedy and trimming down with the score based method.

To our surprise, Figure 3.1 lets us presume that the greedy algorithm works better than the score-based algorithm. We see that for an edge density of 0.1, the score-based method works slightly better. For an edge density of 0.2, they work similarly well, and for all other edge densities, the greedy algorithm works significantly better than the greedy algorithm. The reason for that might be that the more dependencies are apparent in the true underlying distribution of the data, the more likely it becomes that we underfit since the penalizations term becomes too large. Therefore, the best option might be to remove the minimal number of edges, since our model already has too few edges.

3.3 Trimming Initial Guess

The second approach is somewhat similar to the first approach. But this time, instead of trimming G down to a DAG, we use G, to trim down our initial guess $\mathcal{S}^0$. This is done as follows:

$$\mathcal{S}_{ij}^0 = \begin{cases} 1, & \text{if } \mathcal{S}_{ij}^0 = 1 \text{ and } G_{ij} = 1, \\ 0, & \text{else.} \end{cases}$$

This means that we keep only edges in our initial guess S^0 if we have enough evidence for this edge in our bootstrap samples. The difficult part is to choose the correct threshold parameter. Notice that if $t \approx 1$, we would set almost everything to 0, which corresponds to very strong overfitting. At the same time, $t \approx 0$ means that we keep almost every edge which is equivalent to very weak overfitting. But it is, in general, difficult to guess apriori whether we have strong or

weak overfitting. A rule of thumb is that the less data we have, the more likely overfitting might occur. Moreover, $t = 1/2$ usually makes a good job. But in a lot of cases, we have the problem that the initial guess $\mathcal{S}^0$ underfits, which means that we do not find all edges that would explain most of the dependency structure. If we then trim $\mathcal{S}^0$ even further, we most likely will end up with a DAG, which is worse than the initial guess $\mathcal{S}^0$.

3.4 Simulation Studies

In this Section we conduct some simulation studies to see how well the methods presented above work in practice. But before discussing the results, let us introduce some definitions and parameters, which we use for the simulation studies.

In our simulation studies, we always generate some random DAG $\mathcal{S}_t$, which we assume to be the true distribution. Then we sample n data points $\{X_i\}_{i=1}^n$ from $\mathcal{S}_t$. The goal is to use $\{X_i\}_{i=1}^n$ in order to fit a DAG $\mathcal{S}$ such that $\mathcal{S}_t$ is as similar as possible to $\mathcal{S}$. When generating random DAGs, we have to specify the following parameters:

1. `data.dists`: This is a list of distributions. If we work with d-dimensional data, each variable has its proper distribution. Notice that we can choose from Binomial for binary data, Poisson for count data, Gaussian for continuous data, and multinomial for categorical data.

2. `edge.density`: To generate a random DAG we start with the zero matrix $\boldsymbol{M} \in \mathbb{R}^{d\times d}$. Then for all $i > j$, $M_{ij} = 1$, with probability p and $M_{ij} = 0$, with probability $1 - p$. Then $\boldsymbol{M}$ is multiplied by some permutation matrix, such that the vertex ordering changes. The edge density is defined as the above described probability p.

When we fit a model, we always have to specify a method and a score function. The method can either be Bayes or mle. When we choose Bayes, the score function will be the marginal likelihood, or in short, mlik. When choosing mle as our method, we can either choose the BIC or the AIC as the score function. Finally, we need to specify the number of bootstrap samples. In our simulations, we always chose 1000. We start by generating a random DAG with a given edge density. We let the edge density vary between 0.1 and 0.3. We sample a data set $\mathcal{D}$ with 50 data points if the edge density is 0.1, 75 data points if the edge density is 0.2, and 125 data points if the edge density is 0.3. We then use $\mathcal{D}$ to fit a model $\mathcal{S}$ with either mlik, BIC, or AIC as score functions. We use $\mathcal{S}$ as the initial guess for the bootstrapping algorithm, which returns a collection of 1000 DAGs. We use them to construct a summary graph, one time by trimming G down, as described Section 3.1 and the other time trimming $\mathcal{S}$ down as described in Section 3.3. Notice that when trimming G down, we have two algorithms at our disposal. We only discuss the results from the greedy trimming algorithm since, according to the simulation study conducted in 3.2, the greedy algorithm performs better than the score-based algorithm. In addition, we use trimming parameters 0.2, 0.5, and 0.7 for each trimming method.

Lastly, we have to define measures for the goodness of fit. So, we wish to know how close a fitted model is to the true distribution. There are multiple ways of measuring the goodness of fit. Knowing the Markov blanket (1.2) of all vertices, is enough to know all dependencies between all random variables. So it makes sense to put a lot of effort into learning the Markov blanket. To compare models, we can look at the confusion matrices of all Markov blankets.

Notice that there are multiple gray-colored cells in the tables 3.1 - 3.4. These indicate the best fitting models compared to the other trimming algorithms and trimming parameters.

Table 3.1: Comparison of trimming $\mathcal{S}^0$ down and trimming G down for networks with edge density of 0.1. 50 data points are used to fit the models and the data is 10-dimensional. We use the marginal likelihood as the score function.

Simulation	#E	MP		Trim S						Trim G					
				0.2		0.5		0.7		0.2		0.5		0.7	
1	4	6	0	6	0	2	0	0	0	6	0	2	0	0	0
		2	92	2	92	6	92	8	92	2	92	6	92	8	92
2	7	10	22	10	22	6	14	6	6	10	22	6	14	6	6
		8	60	8	60	12	68	12	76	8	60	12	68	12	76
3	4	2	28	2	28	0	0	0	0	2	28	0	0	0	0
		6	64	6	64	8	92	8	92	6	64	8	92	8	92
4	6	16	32	16	32	6	4	0	0	16	24	6	4	0	0
		4	48	4	48	14	76	20	80	4	56	14	76	20	80
5	10	20	42	20	42	2	4	2	4	20	42	2	4	2	4
		10	28	10	28	28	66	28	66	10	28	28	66	28	66

Let us have a look at Table 3.1. Simulation 1 shows that the structure obtained by `mostProbable()` does an excellent job on its own. It has two false negatives, which indicates underfitting, not overfitting. From the confusion matrices of the trimming algorithms, it becomes clear that trimming is not the way to go here.
Simulation 2, however, has 22 false positives. This indicates strong overfitting. At the same time, we only have 8 false negatives. Here bootstrapping might make sense. Note that when trimming $\mathcal{S}^0$ down, trimming parameters 0.5 and 0.7 make an excellent job. They decrease the false positives significantly at the cost of false negatives. When we use 0.7 as our threshold parameter, we lose 16 false positives, while we gain only 4 false negatives, which is a reasonable tradeoff. Trimming G down has the same effect as $\mathcal{S}^0$ down.
The last simulation has a very poor initial fit with 42 false positives, indicating very strong overfitting. So we expect this method to work very well on this fit. Again a parameter of 0.2 has no effect at all. But a trimming parameter of 0.5 decreases the false positives from 42 to 4, which is an exceptional result. At the same time, we only gain 18 false negatives.
Trimming seems to work quite well if the underlying data has low correlation, and in addition, the proposed model indicates to overfit. Surprisingly trimming $\mathcal{S}^0$ down, or trimming G down, seems to make only a difference in simulation 4 when we use a trimming parameter of 0.2. Then trimming G down works significantly better since the number of false positives drops by 6.

Table 3.2: Comparison of trimming $\mathcal{S}^0$ down and trimming G down for networks with edge density of 0.2. 75 data points were used to fit the models and the data was 10-dimensional. Marginal likelihood was used as a score function.

Simulation	#E	MP		Trim S						Trim G					
				0.2		0.5		0.7		0.2		0.5		0.7	
1	8	8	0	8	0	2	0	2	0	8	0	2	0	2	0
		12	80	12	80	18	80	18	80	12	80	18	80	18	80
2	12	18	12	18	12	16	4	6	0	18	12	16	4	6	0
		36	34	36	34	38	42	48	46	36	34	38	42	48	46
3	11	10	12	10	12	0	2	0	0	10	12	0	2	0	0
		22	56	22	56	32	66	32	68	22	56	32	66	32	68
4	8	8	12	6	6	0	0	0	0	6	6	0	0	0	0
		12	68	14	74	20	80	20	80	14	74	20	80	20	80
5	17	10	14	2	2	2	0	0	0	4	2	2	0	0	0
		36	40	44	52	44	54	46	54	42	52	44	54	46	54

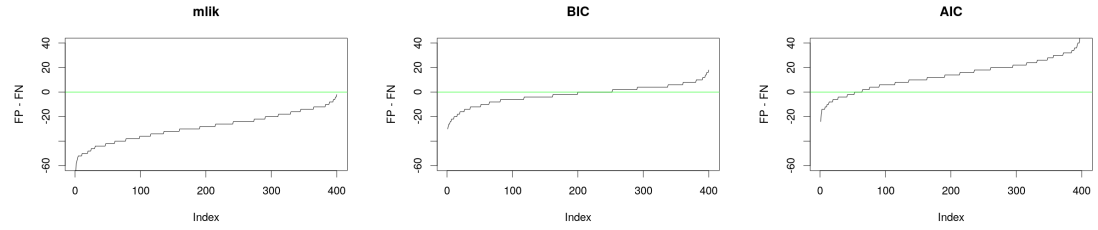
Table 3.2 gives us the following observations. The first simulation is a model that initially underfits, and we have 0 false positives and 12 false negatives. As in the last section, the goodness of fit decreases if we apply trimming to such an initial model. The initial guess of the second simulation, however, has 12 false positives and 36 false negatives. The algorithm, however, should still be able to decrease the number of false positives while having a slight increase in the false positives. If we use 0.5 as our trimming parameter, it seems to work well. We lose 8 false positives by gaining 2 false positives, which is a reasonable tradeoff. By choosing a trimming parameter of 0.7, however, we lose many true positives. So here, a trimming parameter of 0.5 seems reasonable.
The fourth simulation has an initial guess with 12 false positives and 12 false negatives. By applying trimming with threshold parameter 0.2, we lose 6 false positives while gaining 2. But by increasing the threshold parameter even higher, we strongly underfit since they don't even have a single true positive.
The fifth simulation is the first simulation where trimming $\mathcal{S}^0$ down or trimming G down does not result in the same goodness of fit. Trimming G down seems to work slightly better.
To summarize, when applying the bootstrap approach to data sets sampled from a distribution with edge density 0.2 and fitted with score function mlik, the obtained models often underfit instead of overfit. We find a high false positive rate in all the models but an even higher false negative rate. But the trimming approach only decreases the number of false positives at the cost of false negatives. So it might make sense to use a different score function to obtain models that have a higher tendency to overfit than so far.
So let us repeat simulation 1, 3, and 4 from the previous simulation study, seen in Table 3.2 but this time using method mle and score function BIC. Notice that trimming parameter 0.7 is no longer used.

Table 3.3: Comparison between trimming G down and trimming $\mathcal{S}^0$ down when using mlik and BIC as score functions and an edge density of 0.2. 75 data points were used to fit the models.

Simulation	Score	#E	MP		Trim S				Trim G			
					0.2		0.5		0.2		0.5	
1	BIC	8	10	0	8	0	0	0	10	4	0	0
			10	80	12	80	20	80	10	76	20	80
	mlik	8	8	0	8	0	2	0	8	0	2	0
			12	80	12	80	18	80	12	80	18	80
2	BIC	11	18	16	10	4	0	0	12	4	2	0
			14	52	22	64	32	68	20	64	30	68
	mlik	11	10	12	10	12	0	2	10	12	0	2
			22	56	22	56	32	66	22	56	32	66
3	BIC	8	10	12	4	0	0	0	4	0	2	0
			10	68	16	80	20	80	16	80	18	80
	mlik	8	8	12	6	6	0	0	6	6	0	0
			12	68	14	74	20	80	14	74	20	80

Notice that, according to Table 3.3, the BIC scorefunction finds more true positives and true negatives than the mlik score function in all cases. Moreover, BIC has a slightly higher tendency to overfit since the number of false positives is usually slightly larger when using BIC as a score function.
Let us first focus on trimming $\mathcal{S}^0$ down. The first simulation shows that choosing mlik as the score function works better than BIC. But in both cases, it is not advised to apply trimming since we remove true positives while there are no false positives to remove. If we apply a trimming method, we only decrease the number of true positives while increasing the number of false negatives.

In the second simulation, however, choosing BIC to fit the initial model is beneficial when applying trimming to the resulting model. When choosing BIC as a score function, we have 18 true positives while the mlik score function yields only 10. This comes with the cost of more false positives and false negatives in the BIC case. The false positives are not a problem so far since they can be removed using a trimming procedure. Unfortunately, there are more false negatives introduced than false positives. When choosing a trimming parameter of 0.2, the BIC score function removes significantly more false positives than the mlik score function since it has 4 false positives instead of 12 without. The downside is that mlik does not increase the number of false negatives while BIC increases the number by 8.
The same holds for simulation 3. When choosing a trimming parameter of 0.2, using the mlik score function, we still have 6 false positives. Using BIC as a score function, we have 0 while gaining only 2 false negatives compared to the mlik score function.
To summarize, when having an edge density of 0.2, according to our simulations, it might make sense to construct our bootstrap samples, as well as the initial fit, using BIC as a score function, instead of mlik. Moreover, depending on the case, 0.2 or 0.5 as threshold parameters make a lot of sense. But 0.7 is usually too large.
Next, we apply this theory to graphs with edge density 0.3. Before going to the simulations, we want to ensure that we choose the score function wisely. Depending on the score function, it is more or less likely that the initial model overfits. Our theory, however, only works to prevent overfitting, or in other words, to reduce the number of false positives. When the underlying data has a high edge density, the score function becomes crucial. Counter-intuitively, we want our initial model to overfit since our methods would decrease the goodness of fit if it would underfit. BIC, AIC, and mlik all have a regularizer that punishes models with many edges. Therefore the higher the edge density of the true model, the more likely we underfit. Also, the higher the edge density, the stronger the underfitting. Therefore we simulate 400 different data sets of size $n = 125$ and fit models using BIC, AIC, and mlik as score functions to compare how often models overfit or underfit, depending on the score function.



(a) **(b)** **(c)**

Figure 3.2: Plot FP-FN for score functions mlik in (a), BIC in (b) and AIC in (a)

Notice that in 400 simulations, there is not even one simulation where the model has more false positives than false negatives when using mlik as the score function as indicated by (a). The BIC score function has approximately the same number of overfitting models as underfitting models as indicated by (b), and the AIC score function overfits most of the time as indicated by (c). Therefore we use only BIC, and AIC score functions for the upcoming simulations.

Table 3.4: Comparison between trimming G down and trimming $\mathcal{S}^0$ down when using mlik and BIC as score functions. 125 data points were used to fit the models and the data was 10-dimensional.

Simulation	Score	#E	MP		Trim S				Trim G			
					0.2		0.5		0.2		0.5	
1	BIC	15	12	6	10	6	0	0	12	10	0	0
			30	52	32	52	42	58	30	48	42	58
	AIC	15	24	24	22	16	0	2	26	26	0	2
			18	34	20	42	42	56	16	32	42	56
2	BIC	7	16	12	16	12	2	0	12	10	2	0
			4	68	4	68	18	80	8	70	18	80
	AIC	7	14	30	14	30	0	4	18	30	0	2
			6	50	6	50	20	76	2	50	20	76

Notice that, according to Table 3.4, trimming G down or trimming S down results in a worse fit than the initial fit. We used to gain some false negatives when having lower edge densities but lost many more false positives. But with an edge density of 0.3, there are more false negatives introduced than false positives removed. When comparing score functions BIC and AIC, we notice that AIC has more true positives and more false positives. This is because AIC punishes models with higher edge density less than the BIC score function.
In the first simulation, the number of true positives doubles from BIC to AIC score function, but the number of false positives is multiplied by 4, which is a bad tradeoff. The second simulation is even worse. Here, we have approximately the same number of true positives but 2.5 times more false positives. The number of false negatives, however, does not change significantly when comparing BIC and AIC.

From Tables 3.1 - 3.4, we learn that the higher the edge density, the more likely it becomes that using our methods, we start underfitting where our methods that have been discussed so far no longer work. Moreover, mlik works well for networks with a very low edge density, while BIC is recommended for networks with higher edge density since otherwise, the fitted models tend to underfit. AIC so far does not show significant upsides compared to other score functions. One could argue that for very high edge density, mlik and BIC will fit models that underfit dramatically, and hence AIC would make sense in this framework. Therefore the next chapter is dedicated to deal with the issue of high edge density networks.

Chapter 4

Prediction based

4.1 Prediction Algorithm

In Chapter 3, we used a collection of DAGs to summarize them into one single DAG by only keeping the edges that had enough evidence to exist, according to our bootstrap samples. The main issue with this approach is that the resulting DAG no longer overfits but underfits most of the time. Applying bootstrapping is no longer beneficial since we introduce more new false negatives than removing false positives. This is mainly because the higher the edge density of the true distribution, the more likely it becomes that we underfit when applying the function `mostProbable()`. To get models with more edges, one can use a different score function like the AIC or the BIC. One other approach is to use a prediction algorithm that makes predictions on an unseen data set. We then add new edges to the model, depending on which edges seem to improve the prediction accuracy the most. This new model, which contains more edges, can be used, as an initial guess for the trimming procedure, instead of using `mostProbable()` as the initial guess. The advantage is that when applying both algorithms, we first add some new edges to the model, which decreases the number of false negatives. Then the trimming procedure is used to eliminate false positives.

For that purpose, we first need to introduce a prediction algorithm. Notice that we restrict this chapter to binary random variables.
In Definition 1.2, we have seen that the value of Y only depends on $mb(Y, \mathcal{S})$, given that we know the values of $mb(Y, \mathcal{S})$. In order to do predictions of vertex Y, we only have to consider the values of $mb(Y, \mathcal{S})$. Let us assume that $mb(Y, \mathcal{S}) = \{X_1, \ldots, X_k\}$. Then

$$\mathbf{P}(Y = 1 \mid X_1 = x_1, \ldots, X_k = x_k) = \frac{\text{count}(Y = 1, X_1 = x_1, \ldots, X_k = x_k)}{\text{count}(X_1 = x_1, \ldots, X_k = x_k)}.$$

But the problem is that, we might not even have any observations of the form $X_1 = x_1, \ldots, X_k = x_k$. Therefore, we need to take a subset of the Markov blanket. In order to address this issue we proceed in the following way.

We have two data sets. $\mathcal{D}_c$, which is the complete data, which is used to fit the structure and the parameters, and $\mathcal{D}_m$, which is the incomplete data, which needs to be filled. Let us denote

$$X_{-I} = \{X_1 = x_1, \ldots, X_{i_1-1} = x_{i_1-1}, X_{i_1+1} = x_{i_1+1}, \ldots, X_{i_k-1} = x_{i_k-1}, X_{i_k+1} = x_{i_k+1}, \ldots, X_d = x_d\}$$

for $I = \{i_1, \ldots, i_k\}$

Until we find observations in our data set, fulfilling the given constraints, we keep removing a part of the Markov blanket as follows. We start by removing only X_i for $i \leq k$. If there exist

some data points, satisfying $X_1 = x_1, \dots X_{i-1} = x_{i-1}, X_{i+1} = x_{i+1}, \dots X_k = x_k$, then

$$p = \mathbf{P}(Y = 1 \mid X_{-i}) = \frac{\text{count}(Y = 1, X_{-i})}{\text{count}(X_{-i})}$$

and

$$q = \mathbf{P}(Y = 0 \mid X_{-i}) = \frac{\text{count}(Y = 0, X_{-i})}{\text{count}(X_{-i})}.$$

If either $p \neq 0$ or $q \neq 0$, we predict 1 if $p > q$ and 0 otherwise. Let us denote $\mathcal{S}^0$ to be the highest scoring DAG. If there are no observations in the data set, the algorithm removes in each iteration two variables from the Markov blanket. This process is repeated until there is a subset A of the Markov blanket $mb(Y, \mathcal{S}^0)$ such that there exist observations in the data set satisfying the constraints in the variables $mb(Y, \mathcal{S}^0) \setminus A$. We then predict all missing values in $\mathcal{D}_m$. The filled data set is denoted by $\mathcal{D}_f$. We then define $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_f$ and $\mathcal{D}$ to recompute the probabilities and reassign the missing values of $\mathcal{D}_m$. We reiterate this process until the predictions do not change anymore. R-Code 4.1 is an implementation of this algorithm.

R-Code 4.1. Prediction algorithm.

```
increment <- function(v) {
  l <- length(v)
  if (l == 1) {
    return(c(1))
  }
  for (i in l:2) {
    if (v[i] == 1 & v[i-1] == 0) {
      v[i] <- 0
      v[i-1] <- 1
      return(v)
    }
  }
  s <- sum(v) + 1
  v <- c(rep(0, l-s), rep(1, s))
}

constructList <- function(data, dag, dists) {
  d <- dim(dag)[1]
  n <- dim(data)[1]
  names <- names(dists)
  fulls <- list()
  conds <- list()
  probs <- list()
  for (i in 1:d) {
    data.int <- sapply(data[,], as.integer)-1
    markov.blanket <- mb(dag, names[i], dists)
    if (is.null(markov.blanket)) {
      p <- (n - sum(as.numeric(data[,names[i]])-1))/n
      q <- (sum(as.numeric(data[,names[i]])-1))/n
      fulls[[i]] <- c(p, q)
      probs[[i]] <- c(p, q)
    }
    else {
```

```
      name.indices <- which(names %in% markov.blanket)
      data.full <- as.data.frame(data.int[,c(i, name.indices)])
      data.partial <- as.data.frame(data.int[,c(name.indices)])
      full <- rep(0, 2^(length(markov.blanket)+1))
      cond <- rep(0, 2^(length(markov.blanket)))

      for (k in 1:nrow(data.full)) {
        ind <- 0
        for (l in 1:ncol(data.full)) {
          ind <- ind + data.full[k, ncol(data.full)-l+1]*2^(l-1)
        }
        full[ind+1] <- full[ind+1] + 1
      }

      for (k in 1:nrow(data.partial)) {
        ind <- 0
        for (l in 1:ncol(data.partial)) {
          ind <- ind + data.full[k, ncol(data.full)-l+1]*2^(l-1)
        }
        cond[ind+1] <- cond[ind+1] + 1
      }

      fulls[[i]] <- full
      conds[[i]] <- cond

      p <- rep(0, length(fulls[[i]]))
      for (m in 1:(length(fulls[[i]])/2)) {
        if (cond[m] == 0) {
          p[m] <- 0
        } else {
          p[m] <- full[m]/cond[m]
          p[m+2^length(markov.blanket)] <- full[m+2^length(markov.blanket)]/cond[m]
        }
      }
      probs[[i]] <- p
    }
  }
  return(probs)
}

probability <- function(probs, vertex, markov.blanket, keep, datapoint) {
  para <- probs[[node]]
  m <- length(markov.blanket)
  countNA <- sum(keep)
  comb <- rep(0, countNA)
  p <- 0
  q <- 0
  if (countNA != 0) {
    for (i in 1:2^countNA) {
```

```
      conditinal <- keep
      cnt <- 1
      for (j in 1:m) {
        if (conditinal[j] == 0) {
          conditinal[j] <- datapoint[j]
        } else {
          conditinal[j] <- comb[cnt]
          cnt <- cnt + 1
        }
      }
      ind <- 0
      for (j in 1:m) {
        ind <- ind + conditinal[length(conditinal)-j+1]*2^(j-1)
      }
      p <- p + para[ind + 1]
      q <- q + para[ind + 2^m + 1]
      if (!identical(comb, rep(1, countNA))) {
        comb <- increment(comb)
      }
    }
  } else {
    ind <- 0
    for (j in 1:m) {
      ind <- ind + datapoint[length(datapoint)-j+1]*2^(j-1)
    }
    p <- para[ind + 1]
    q <- para[ind + 2^m + 1]
  }
  return(c(p, q))
}

predict.single <- function(probs, node, markov.blanket, datapoint) {
  m <- length(markov.blanket)
  keep <- rep(0, m)
  p <- 0
  q <- 0
  if (!is.null(markov.blanket)) {
    while ((p == 0 & q == 0)|(p == 0.5 & q == 0.5)) {
      v <- probability(probs, node, markov.blanket, keep, datapoint)
      p <- v[1]
      q <- v[2]
      if (!identical(keep, rep(1, m))) {
        keep <- increment(keep)
      }
    }
  } else {
    if (probs[[node]][1] >= probs[[node]][2]) {
      return(0)
    } else {
      return(1)
```

```
    }
  }
  if (p > q) {
    return(0)
  } else {
    return(1)
  }
}

EM <- function(data.complete, data.incomplete, dag, dists) {
  data.all <- data.complete
  data.incomplete.keep <- data.incomplete
  n <- dim(data.incomplete)[1]
  res.new <- rep(0, n)
  res.old <- rep(NA, n)
  n <- dim(data.incomplete)[1]
  names <- names(dists)
  while (!identical(res.new, res.old)) {
    res.old <- res.new
    res.new <- c()
    probs <- constructList(data.all, dag, dists)
    for (i in 1:n) {
      ind <- which(is.na(data.incomplete[i, ]))
      markov.blanket <- mb(dag, names[ind], dists)
      datapoint <- as.numeric(as.vector(unlist(
        data.test[i, which(names %in% markov.blanket)])))
      pred <- predict.single(probs, ind, markov.blanket, datapoint)
      data.incomplete[i, ind] <- as.factor(pred)
      res.new <- c(res.new, pred)
    }
    data.all <- rbind(data.complete, data.incomplete)
    data.incomplete <- data.incomplete.keep
  }
  return(res.new)
}
```

4.2 Estimating Markov Blanket

In Chapter 2, we used the bootstrap samples to find confidence bands for the edges in our graph. Then only the edges that had enough evidence, according to the bootstrap samples, were kept. Here we chose a slightly different approach. Since we have a prediction algorithm, we can test how well each bootstrap sample performs and compare them by prediction accuracy. The problem with the mlik or BIC is that for data with many dependencies, we often underfit. But by measuring the prediction accuracy, we neither over- nor underfit. So there is potential that we can deal with this issue.

In the following, we introduce an algorithm that takes as input some initial guess $\mathcal{S}^0$ of the structure and our bootstrap samples and returns a DAG $\mathcal{S}^*$ that hopefully better fits the data.

R-Code 4.2 introduces an algorithm, that constructs an accuracy matrix $\boldsymbol{A} \in \mathbb{R}^{N \times d}$ where N is

the number of bootstrap samples, and d is the number of variables per data point. A_{ij} is defined to be the accuracy of bootstrap sample i for predicting variable j given all other variables.

R-Code 4.2. Construct accuracy matrix.

```
constructAccuracyMatrix <- function(data.train, data.test, boot, dists) {
  accuracies <- matrix(rep(0, length(boot)*dim(data.test)[2]), nrow = length(boot))
  n <- dim(data.test)[1]
  d <- dim(data.test)[2]
  N <- length(boot)
  data.test.keep <- data.test
  for (i in 1:N) {
    for (j in 1:d) {
      data.test <- data.test.keep
      true <- rep(0, n)
      for (k in 1:n) {
        true[k] <- data.test[k, j]
        data.test[k, j] <- NA
      }
      true <- true - 1
      pred <- EM(data.train, data.test, boot[[i]], dists)
      acc <- 1-sum(abs(pred-true))/n
      accuracies[i,j] <- acc
    }
  }
  return(accuracies)
}
```

At the same time, we define $a \in \mathbb{R}^d$ such that a_i is the predictive accuracy of our initial guess $\mathcal{S}^0$ for predicting variable i given all other variables. We implemented this function in R-Code 4.3.

R-Code 4.3. Construct accuracy matrix for the initial guess $\mathcal{S}^0$.

```
predict.mp <- function(mp.dag, data.train, data.test, dists) {
  data.test.keep <- data.test
  d <- dim(data.train)[2]
  accuracies.mp <- rep(0, d)
  n <- dim(data.test)[1]
  for (j in 1:d) {
    data.test <- data.test.keep
    true <- rep(0, n)
    for (k in 1:n) {
      true[k] <- data.test[k, j]
      data.test[k, j] <- NA
    }
    true <- true - 1
    pred <- EM(data.train, data.test, mp.dag, dists)
    acc <- 1-sum(abs(pred-true))/n
    accuracies.mp[j] <- acc
  }
```

```
  return(accuracies.mp)
}
```

In R-Code 4.4, we define $\boldsymbol{M} \in \mathbb{R}^d$ such that $M_i = \max_{k \leq N} A_{ki}$. In words M_i, is the best predictive performance for vertex i out of all N bootstrap samples. Then pick $j = \arg\max_{i \leq d} M_i - a_i$. In words j, is the vertex that has the most potential for improving its predictive performance.

R-Code 4.4. Construct a list of all possible edges that could be added to *mp.dag* such that the Markov blanket of the vertex in question has at least one of the vertices specified by *mb* in its Markov blanket.

```
returnPossibleEdges <- function(mp.dag, node, mb) {
  nodes <- matrix(c(0,0), nrow = 1)
  d <- dim(mp.dag)[1]
  for (i in 1:length(mb)) {
    tmp <- mp.dag
    tmp[mb[i], node] <- 1
    if (mp.dag[mb[i], node] == 0 &
        length(return_cycles(tmp)) == 0 & sum(diag(tmp)) == 0) {
      nodes <- rbind(nodes, c(node, mb[i]))
    }
  }

  for (i in 1:length(mb)) {
    tmp <- mp.dag
    tmp[mb[i], node] <- 1
    if (mp.dag[node, mb[i]] == 0 &
        length(return_cycles(tmp)) == 0 & sum(diag(tmp)) == 0) {
      nodes <- rbind(nodes, c(mb[i], node))
    }
  }

  children <- which(mp.dag[,node] == 1)
  parents <- which(mp.dag[node,] == 1)

  for (i in 1:length(mb)) {
    for (j in 1:d) {
      tmp <- mp.dag
      tmp[mb[i], node] <- 1
      if (j %in% children & mp.dag[mb[i], j] == 0 &
          length(return_cycles(tmp)) == 0 & sum(diag(tmp)) == 0) {
        nodes <- rbind(nodes, c(mb[i], j))
      }
    }
  }

  for (i in 1:length(mb)) {
    markov.child <- which(mp.dag[,mb[i]] == 1)
    if (length(markov.child) >= 1) {
```

```
      for (j in 1:length(markov.child)) {
        tmp <- mp.dag
        tmp[mb[i], node] <- 1
        if (mp.dag[markov.child[j], node] == 0 &
            length(return_cycles(tmp)) == 0 & sum(diag(tmp)) == 0) {
          nodes <- rbind(nodes, c(node, markov.child[j]))
        }
      }
    }
  }
  nodes <- nodes[-1, ]
  nodes <- matrix(nodes, ncol = 2)
  return(nodes)
}
```

We define $\mathcal{B}$ to be the DAG in our bootstrap sample that has the highest predictive performance on vertex j. We compute the Markov blanket $mb(X_j, \mathcal{B})$. This Markov blanket is likely to contain most of the variables that are also part of the Markov blanket of X_j of the true distribution. This is because the value of X_j only depends on the value of its Markov blanket. Since the DAG performs so well, we can most likely find the most relevant dependencies of X_j in $\mathcal{B}$. Then we construct a list of all edges that can be added to $\mathcal{S}^0$ such that the resulting graph still does not contain any cycles and additionally satisfies that $mb(X_j, \mathcal{S}^0)$ contains at least one more vertex which is contained in $mb(X_j, \mathcal{B})$.

We add each of the edges, which we found in the previous step to structure $\mathcal{S}^0$, compute the predictive performance, and remove again the vertex. This process is repeated until every possible edge that could have been added has been tried out. Then the edge, that improves the predictive performance the most is being added to $\mathcal{S}^0$. If there the prediction accuracy drops, no matter which edge we insert, the given vertex is added to a blacklist, meaning that this variable is no longer considered for the upcoming iterations. We do not add an edge if this occurs. This algorithm is implemented in R-Code 4.5

R-Code 4.5. Iteratively adds edges returned by the function *returnPossibleEdges()* and computes the predictive accuracy of *mp.dag* if the given vertices are added.

```
addEdges <- function(mp.dag, names, accuracies, boot, data.train,
                     data.test, max.iter = 10, blanket = NULL) {
  cnt <- 0
  d <- length(names)
  ignore <- c()
  max.acc <- apply(accuracies, 2, max)
  mp.predict <- predict.mp(mp.dag, data.train, data.test, dists)
  addedArcs <- matrix(c(0,0), nrow = 1)
  dags <- list()
  accuraciesPerNode <- matrix(mp.predict, nrow = 1)
  accuracy.diff <- max.acc-mp.predict
  print(mp.predict)
  while (cnt <= max.iter) {
    cnt <- cnt + 1
```

```
    set.zero <- c()
    edges <- c()
    while (length(set.zero)<d & length(edges) == 0) {
      accuracy.diff[set.zero] <- 0
      accuracy.diff[ignore] <- 0
      node <- which.max(accuracy.diff)
      k <- which.max(accuracies[,node])
      if (is.null(blanket)) {
        markov.blanket <- which(names %in% mb(boot[[k]], names[node], dists))
      } else {
        markov.blanket <- blanket[[node]]
      }
      edges <- returnPossibleEdges(mp.dag, node, markov.blanket)
      if (length(edges) == 0) {
        set.zero <- c(set.zero, node)
      }
    }

    acc <- c()
    print(dim(edges)[1])
    for (i in 1:dim(edges)[1]) {
      print(i)
      mp.dag[edges[i,2], edges[i,1]] <- 1
      if (length(return_cycles(mp.dag)) >= 1 | sum(diag(mp.dag))>0) {
        mp.dag[edges[i,2], edges[i,1]] <- 0
        acc <- c(acc, NA)
      } else {
        mp <- predict.mp(mp.dag, data.train, data.test, dists)
        acc <- c(acc, sum(mp))
        mp.dag[edges[i,2], edges[i,1]] <- 0
      }
    }
    ind <- which.max(acc)
    mp.dag[edges[ind, 2], edges[ind, 1]] <- 1
    mp.predict <- predict.mp(mp.dag, data.train, data.test, dists)
    addedArcs <- rbind(addedArcs, c(edges[ind, 2], edges[ind, 1]))
    accuraciesPerNode <- rbind(accuraciesPerNode, mp.predict)
    dags[[cnt]] <- mp.dag
    accuracy.diff <- max.acc-mp.predict
    print(mp.dag)
    print(mp.predict)
    if (all(names[blanket[[node]]] %in% mb(mp.dag, names[node], dists))) {
      ignore <- c(ignore, node)
    }
  }
  return(list(accuraciesPerNode, addedArcs, dags))
}
```

This entire process is repeated multiple times. If we repeat it too many times, we will overfit since, at each step, we add new edges. So to deal with this issue, it might make sense to repeat

that process a reasonable number of times and store the predictive performances after every iteration step. Then we plot the predictive accuracies per iteration step and evaluate when the predictive performance decreases. The function itself returns for each iteration step a vector that contains the predictive performances per vertex and the corresponding DAG. It remains to choose the appropriate DAG out of this collection of DAGs.

R-Code 4.6. Computes the confusion matrix.

```
confusionMatrix <- function(true.dag, my.dag, dists) {
  full <- matrix(rep(0, 4), nrow = 2)
  d <- dim(true.dag)[1]
  names <- names(dists)
  for (i in 1:d) {
    mb.true <- mb(true.dag, names[i], dists)
    mb.my <- mb(my.dag, names[i], dists)
    for (j in 1:length(names)) {
      if (names[j] %in% mb.my & names[j] %in% mb.true) {
        full[1,1] <- full[1,1] + 1
      } else if (names[j] %in% mb.my & !(names[j] %in% mb.true)) {
        full[1,2] <- full[1,2] + 1
      } else if (!(names[j] %in% mb.my) & names[j] %in% mb.true) {
        full[2,1] <- full[2,1] + 1
      } else {
        full[2,2] <- full[2,2] + 1
      }
    }
  }
  return(full)
}
```

4.3 Example

Before going to the simulation study to see how this method works in general, it makes sense to go through an example, step by step. Let us generate a DAG with edge density 0.4. We assume the following DAG:

R-Code 4.7. Defining distributions, generating DAG.

```
names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')

set.seed(8)
true.dag <- simulateDag(names, dists, 0.4)$dag
```

Then we generate $n = 100$ data points to fit an ABN using the *mostProbable()* command.

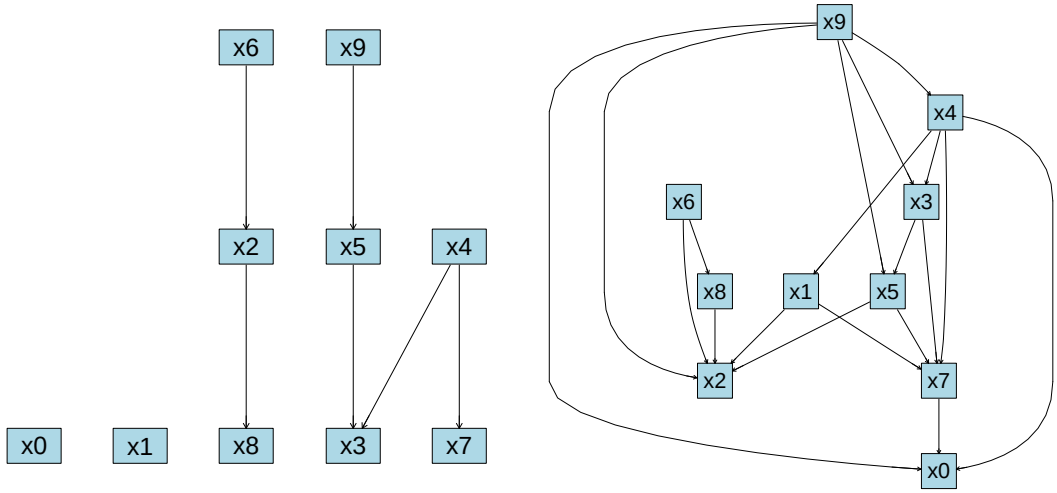
R-Code 4.8. Sample data, compute highest-scoring DAG.

```
data.train <- simulateAbn(data.dists = dists, data.param = true.dag,
                          n.adapt = 1000, n.chains = 1, n.thin = 10,
                          n.iter = 1000, seed = 4)

data.test <- simulateAbn(data.dists = dists, data.param = true.dag,
                         n.adapt = 1000, n.chains = 1, n.thin = 10,
                         n.iter = 1000, seed = 5)

mycache <- buildScoreCache(data.df = data.train, data.dists = dists,
                           method = 'bayes', max.parents = 8)

mp.dag <- mostProbable(score.cache = mycache)
mp.dag <- mp.dag$dag
```



(a) **(b)**

Figure 4.1: Plots of the DAGs that represents the dependencies of the underlying data and the DAG obtained by applying the function *mostProbable()*

Notice that the confusion matrix of the highest-scoring DAG (Figure 4.1b) is

$$\begin{bmatrix} 14 & 0 \\ 46 & 40 \end{bmatrix}.$$

The highest-scoring DAG (Figure 4.1b) dramatically underfits the true DAG. Therefore, it makes sense to add some edges to the highest-scoring network. Let us have a look at the Markov blankets and see how well the highest-scoring DAG estimates the Markov blankets.
Overall, we see that there are 14 true positives and 40 true negatives. Moreover, there are no false positives, which is not surprising since we conservatively added edges. At the same time, there are 40 false negatives which indicates strong underfitting.

R-Code 4.9. Generate 1000 bootstrap samples, compute accuracies of all bootstrap samples, add most probable edges.

```
boot <- bootstrap(num.of.simulations = 1000, data = data, method = 'bayes',
                  score = 'mlik', dists = dists, max.par = 8, ban = NULL,
                  retain = NULL, n.chains = 1, n.iter = 1000,
                  n.adapt = 1000, n.thin = 10)
accuracies <- constructAccuracyMatrix(data.train = data.train,
                                      data.test = data.test,
                                      boot = boot, dists = dists)
addedEdges <- addEdges(mp.dag, names, accuracies, boot, data.train, data.test)
```

In R-Code 4.9, we apply the procedure described in Section 4.2 to the data sampled in R-Code 4.8. We can look at the prediction accuracy plots over time, to see which is the optimal time to stop adding additional edges:

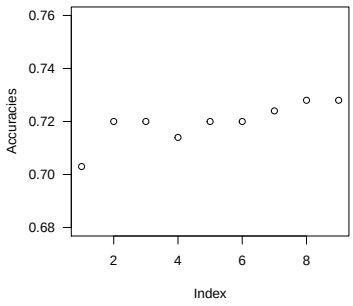


Figure 4.2: Plot the mean accuracy for each iteration step of the algorithm introduced in Chapter 4.2.

From Figure 4.2 we can deduce that the predictive performance no longer improves after iteration 8.

Let us now see the confusion matrix for the Markov blankets for the hopefully better performing DAG:

$$\begin{bmatrix} 22 & 6 \\ 38 & 34 \end{bmatrix}.$$

The number of true positives increases by 8 while the number of false negatives decreases by 8 as well. On the other hand, the number of false positives increases by 6, and true negatives decrease by 6. Therefore this particular example seems to work not too well. Let us propose an alternative approach.

4.4 Alternative Approach

In this section, we introduce an alternative algorithm to the one proposed in Section 4.2. The disadvantage of the so far used algorithm is that we only consider the best predicting DAG. We use the optimal DAG for guessing the optimal Markov blankets.

Here, for each vertex i, we consider a collection of DAGs $G_i = \{G_i^j\}_{j=1}^{k_i}$, namely the set of all DAGs that have an accuracy greater or equal than some user-defined threshold. We define a matrix $\boldsymbol{M} \in \mathbb{R}^{d\times d}$ such that

$$M_{ij} = \sum_{l=1}^{k_i} \mathbf{1}_{X_j \in mb(G_i^l)}. \tag{4.1}$$

In other words, M_{ij} is the number of times that vertex j appears in a Markov blanket of G_i. Then for each vertex, the user decides how many vertices are assumed in each Markov blanket. So, for example, if for vertex X_1, the user chooses to use the four most relevant vertices, the algorithm will pick the four vertices that have the most evidence to be in the Markov blanket of X_1 according to $\boldsymbol{M}$. Then the same procedure is repeated as in the algorithm in Section 4.2.

The disadvantage of this algorithm is that not everything is fully automated. The user has to decide the values of the accuracy threshold for each vertex and how many vertices should be in each Markov blanket. At the same time, this can be an advantage. We could have some prior knowledge on how dense the resulting network should be, and then we can encompass this prior knowledge here. The algorithm introduced in Section 4.2 can easily be generalized using the helper functions in R-Code 4.10.

R-Code 4.10. Helper functions.

```
plotAccuracies <- function(accuracies) {
  d <- dim(accuracies)[2]
  for (k in 1:d) {
    plot(accuracies[,k], ylab = 'Accuracy',
         xlab = 'Index', ylab  =  "", yaxt = "n")
  }
}

countMb <- function(accuracies, thresholds, boot, dists) {
  names <- names(dists)
  d <- dim(accuracies)[2]
  count <- matrix(rep(0, d^2), nrow = d)
  for (i in 1:d) {
    ind <- which(accuracies[,i] >= thresholds)
    for (j in 1:length(ind)) {
      dag <- boot[[ind[j]]]
      markov.blanet <- mb(dag, names[i], dists)
      tmp <- names %in% markov.blanet
      if (is.null(tmp)) {
        tmp <- rep(0, d)
      }
      count[i, ] <- count[i, ] + tmp
    }
  }
  return(count)
}

returnMb <- function(count, numberPerNodes) {
  d <- dim(count)[1]
  blanket <- list()
```

```
  for (i in 1:d) {
    blanket[[i]] <- which(count[i,] %in% sort(count[i,], decreasing = T)
                          [1:numberPerNodes[i]])
  }
  return(blanket)
}
```

4.5 Example

In this section, we use the same DAG and bootstrap sample as in the example of Section 4.5. However, this time we will proceed with the second approach to see which of both methods work better.

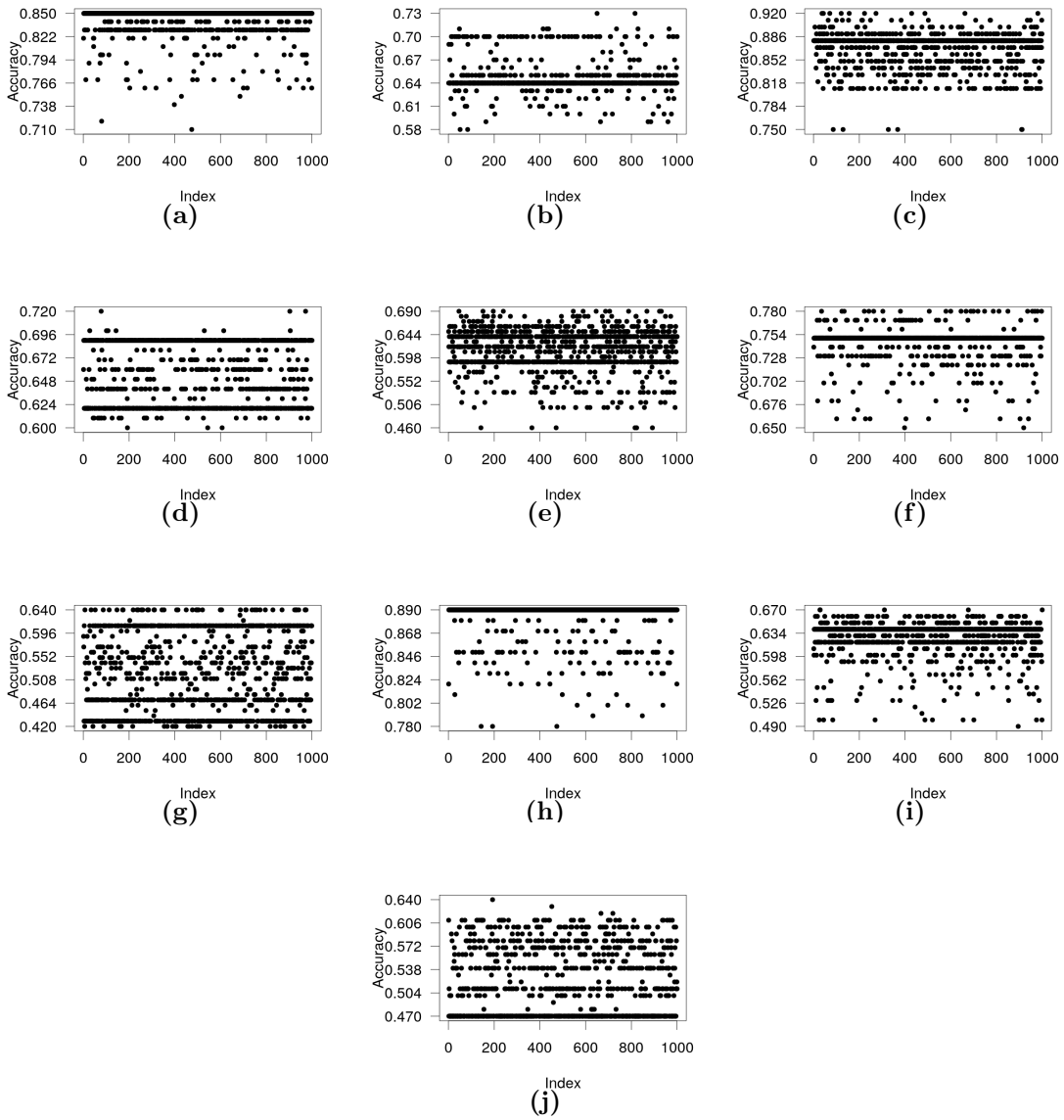


Figure 4.3: Each plot represents the accuracies for a vertex of all bootstrap samples using the DAG from Figure 4.1b, as the initial structure $\mathcal{S}^0$.

Given Figure 4.3, the user's job is to choose a good value for the minimum accuracy per variable. In this example, we chose the minimum as in R-Code 4.11:

R-Code 4.11. Define minimal accuracies to consider.

```
acc.threshold <- c(0.85, 0.7, 0.9, 0.69, 0.66, 0.76, 0.61, 0.89, 0.65, 0.6)
```

In R-Code 4.12, we construct the count matrix $\boldsymbol{M}$ as defined in equation (4.1)

R-Code 4.12. Construct matrix from Equation 4.1

```
count <- countMb(accuracies, acc.threshold, boot, dists)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0   55  349   94  216  106  146   82  213   147
##  [2,]   12    0   70   87  204   11   23   54   23    16
##  [3,]  377  176    0  207  674  189  448  132  645   400
##  [4,]   41   86   85    0  125   28   39   41   52    37
##  [5,]   62  124  125  117    0   25   40   58   91    52
##  [6,]   87   32  146   46  120    0   95   75  112    95
##  [7,]    6    2   18    7    5    5    0    9   56     8
##  [8,]   91  163  133  137  228  103  161    0  189   172
##  [9,]   61   10  147   27  108   30  108   53    0    73
## [10,]    8    0   12    5   10   11    8    7   12     0
```

Given this matrix, it is not apparent how to chose the number of relevant Markov blanket members per vertex. Usually, it is best to select a larger number per vertex if more dependencies in the variables are apparent. But we do not necessarily know that at the beginning. In principle, we can try out different values and check the predictive performances on an unseen test set. Then one can pick the combination that has the highest prediction accuracy. In this example, we chose it according to R-Code 4.13.

R-Code 4.13. Define the number of vertices that are considered to be added to the Markov blanket.

```
num <- c(5, 1, 5, 2, 3, 3, 1, 5, 3, 4)
```

In R-Code 4.14, we repeat the same procedure as in the algorithm proposed in Section 4.5 with the difference that the Markov blankets that we assume to be the correct ones are different ones.

R-Code 4.14. Define a possible set of vertices for each vertex and add 10 vertices to the model.

```
markov.blanket <- returnMb(count, num)

addedEdges <- addEdges(mp.dag, names, accuracies, boot, data.train, data.test,
                       10, markov.blanket)
```

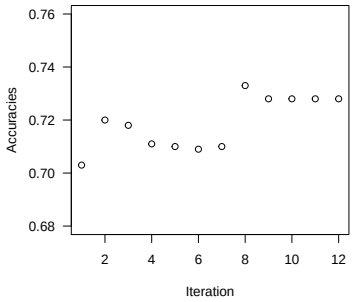
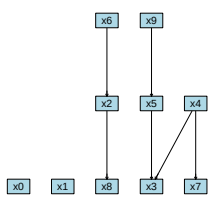
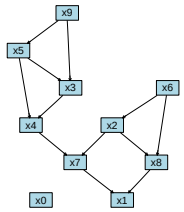
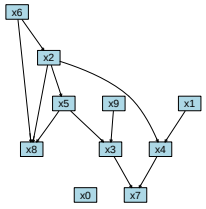


Figure 4.4: Plot the mean accuracy for each iteration step of the algorithm introduced in Chapter 4.2.

From Figure 4.4 we can deduce that the predictive performance no longer improves after iteration 8. Let us now see the confusion matrix for the Markov blankets for the hopefully better performing DAG.

Let us now see the confusion matrix for the Markov blankets for the hopefully better-performing DAG:

Table 4.1: Comparison of the confusion matrices of the DAG obtained by applying the function `mostProbable()`, the algorithm introduced in Section 4.2 and the algorithm introduced in Section 4.4.

`mostProbable()`	Approach 1	Approach 2
$\begin{bmatrix} 14 & 0 \\ 46 & 40 \end{bmatrix}$	$\begin{bmatrix} 22 & 6 \\ 38 & 34 \end{bmatrix}$	$\begin{bmatrix} 28 & 2 \\ 32 & 38 \end{bmatrix}$
		

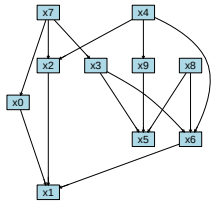
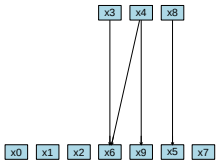
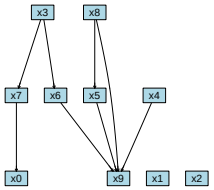
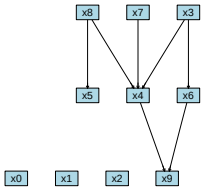
Notice that, according to Table 4.1 the fit obtained by `mostProbable()` strongly underfits the true DAG. It has 46 false negatives. Approach 1 can decrease the number of false positives to 38 at the cost of gaining 6 false negatives. Also, the number of true negatives decreases by 6. We accumulate additional 8 true positives. We can say that it is for sure an improvement since we reconstruct many of the missing dependencies. But unfortunately, using approach 1, we add too many false dependencies to the structure. Approach 2 only increases the number of false positives by 2 while decreasing the number of false negatives by 14, which is extraordinary. Also, we increased the number of true positives by 14 while only losing 2 true negatives. Therefore it might seem like approach 2 dominates approach 1 in the sense that, in general, it performs better. Let us conduct a simulation study to see if this might be true in general.

4.6 Simulation Study

In this simulation study, we want to see how well the two approaches help to decrease the number of false negatives in the Markov blankets. To do so we conduct 5 simulations, in each of which

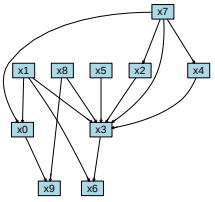
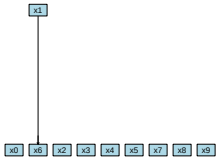
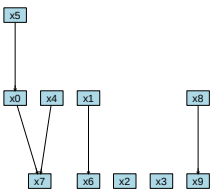
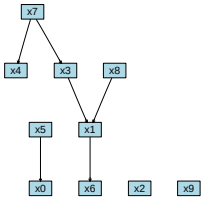
we generate a DAG with edge density 0.3. From each of them, we sample two data sets of size 100. $\mathcal{D}_{\text{train}}$ to fit the initial structure $\mathcal{S}^0$ using the function `mostProbable()` and $\mathcal{D}_{\text{test}}$ as our test set for prediction purposes. We use $\mathcal{S}^0$ to improve the initial fit using the approaches described in Chapter 4.2 and Chapter 4.4.

Table 4.2: Comparison confusion matrices and DAGs, obtained by `mostProbable()`, algorithm introduced in Section 4.2 and the algorithm introduced in Section 4.4, with edge density 0.3 and $n = 125$ data points to fit the model.

True	`mostProbable()`	Approach 1	Approach 2
	$\begin{bmatrix} 10 & 0 \\ 36 & 54 \end{bmatrix}$	$\begin{bmatrix} 20 & 6 \\ 26 & 48 \end{bmatrix}$	$\begin{bmatrix} 18 & 4 \\ 28 & 50 \end{bmatrix}$
			

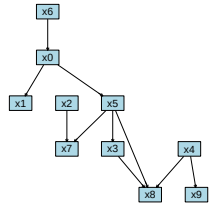
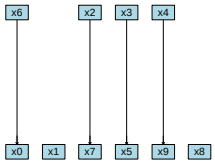
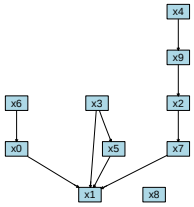
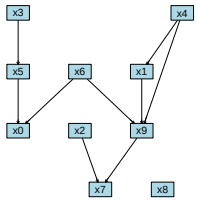
According to Table 4.2, the DAG obtained by Approach 1 gains 10 true positives while losing 6 true negatives and introducing 6 new false positives. These should not be a problem since the trimming procedure is likely to remove them again. Moreover, it indicates that two variables are independent of the remaining random variables, while in the DAG obtained by Approach 2, 3 are. Additionally, the resulting DAG of Approach 1 has 1 more edge than approach 2. Therefore the first simulation seems to indicate that Approach 1 is less conservative in adding new edges. In this case, the resulting DAG from Approach 1 and Approach 2 have similar goodness of fit.

Table 4.3: Comparison of confusion matrices and DAGs, obtained by `mostProbable()`, algorithm introduced in Section 4.2 and the algorithm introduced in Section 4.4, with edge density 0.3 and $n = 125$ data points to fit the model.

True	`mostProbable()`	Approach 1	Approach 2
	$\begin{bmatrix} 2 & 0 \\ 54 & 44 \end{bmatrix}$	$\begin{bmatrix} 8 & 4 \\ 48 & 40 \end{bmatrix}$	$\begin{bmatrix} 12 & 2 \\ 44 & 42 \end{bmatrix}$
			

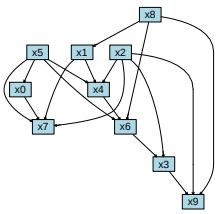
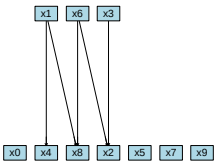
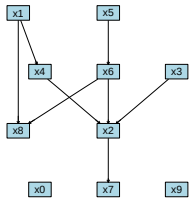
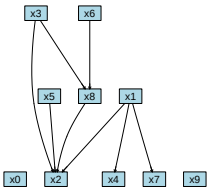
According to Table 4.3, the number of true positives increases by 6 or 10 depending on which approach we chose. At the same time, we decrease the number of true negatives by 4 or 2, respectively. Therefore, the algorithms do an excellent job only adding edges to the graph, which seems to improve the goodness of fit. At the same time, we gain almost no false negatives, which means that trimming is unnecessary since, by trimming, we would introduce more false negatives that we would lose false positives. The reason why both approaches work so well is that the function `mostProbable()` does a bad job, in the sense that it finds by far not all dependencies. In this case, Approach 2 works significantly better than Approach 1.

Table 4.4: Comparison of confusion matrices and DAGs, obtained by `mostProbable()`, algorithm introduced in Section 4.2 and the algorithm introduced in Section 4.4, with edge density 0.3 and $n = 100$ data points to fit the model.

True	`mostProbable()`	Approach 1	Approach 2
	$\begin{bmatrix} 8 & 0 \\ 18 & 74 \end{bmatrix}$	$\begin{bmatrix} 14 & 14 \\ 12 & 60 \end{bmatrix}$	$\begin{bmatrix} 10 & 16 \\ 16 & 58 \end{bmatrix}$
			

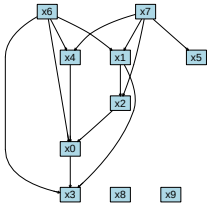
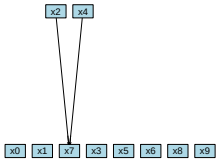
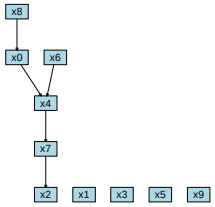
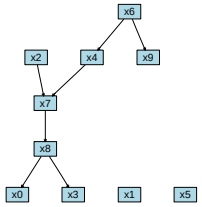
According to Table 4.4 the function `mostProbable()` does an excellent job in finding the initial structure $\mathcal{S}^0$ for this simulation. Therefore both approaches significantly decrease the goodness of fit. Introducing new edges in the graph increases the number of true positives significantly but at the same time increases the number of false negatives and false positives even stronger. One can argue that we can trim those new structures as discussed in Chapter 3. But the problem with that is that it is computationally demanding. The trimming procedure might worsen the fit by removing too many dependencies in the true distribution.

Table 4.5: Comparison of confusion matrices and DAGs, obtained by `mostProbable()`, algorithm introduced in Section 4.2 and the algorithm introduced in Section 4.4, with edge density 0.3 and $n = 100$ data points to fit the model.

True	`mostProbable()`	Approach 1	Approach 2
	$\begin{bmatrix} 12 & 2 \\ 42 & 44 \end{bmatrix}$	$\begin{bmatrix} 20 & 4 \\ 34 & 42 \end{bmatrix}$	$\begin{bmatrix} 24 & 4 \\ 30 & 42 \end{bmatrix}$
			

The DAG obtained by `mostProbable()` in Table 4.5, is an example of a DAG that underfits the true relationships in the data. Both approaches increase the number of true positives by 8 and 12 respectively, introducing 2 new false positives, which is an excellent tradeoff. Also, the number of true negatives does not decrease significantly. Both approaches consider variables x_0 and x_9 to be independent of all other variables. The true distribution, however, considers both of these variables to contain many dependencies. This might be because the initial structure obtained by `mostProbable()` considers x_0 and x_9 to be independent of all the other variables. The bootstrap algorithm constructs DAGs in a neighborhood of the initial structure. Therefore getting edges between those initially disconnected vertices and the rest of the DAG is unlikely. Moreover, according to the true distribution, $mb(x_0, \mathcal{S}_{\text{true}}) = \{x_1, x_2, x_5, x_7\}$. The structure obtained by `mostProbable()` however, considers x_5, x_7 to be independent of all other variables. Therefore, a huge part of the Markov blanket of x_0 is disconnected from all other variables in the initial structure. Hence, the bootstrapping approach is unlikely to obtain enough bootstrap samples, indicating that x_5 and x_7 are part of the Markov blanket of x_0.

Table 4.6: Comparison of confusion matrices and DAGs, obtained by `mostProbable()`, algorithm introduced in Section 4.2 and the algorithm introduced in Section 4.4, with edge density 0.3 and $n = 100$ data points to fit the model.

True	`mostProbable()`	Approach 1	Approach 2
	$\begin{bmatrix} 6 & 0 \\ 28 & 66 \end{bmatrix}$	$\begin{bmatrix} 10 & 2 \\ 24 & 64 \end{bmatrix}$	$\begin{bmatrix} 8 & 8 \\ 26 & 58 \end{bmatrix}$
			

The initial structure from Table 4.6 dramatically underfits the true distribution. Also, x_8 and x_9 do not have any dependencies. The DAG obtained by `mostProbable()` correctly classifies those two variables as independent from all the other variables. But at the same time, it considers many other variables to be independent of all other variables, which in truth are not. Therefore, applying those algorithms is necessary. Approach 1 returns a DAG that still has 4 variables which seem to be independent of all other variables, and unfortunately, x_8 is none of them. Notably, x_6 is independent of all other variables in the DAG obtained by Approach 1, but x_6 has many dependencies. The DAG obtained by Approach 2 only has 2 variables independent from all other variables, and unfortunately, x_8 and x_9 are none of them.

Chapter 5

Discussion and Outlook

In this work, we started by introducing a bootstrap algorithm that computed some initial high-scoring DAG $\mathcal{S}^0$ and sampled N data sets from $\mathcal{S}^0$. Those data sets were used to fit N corresponding DAGs. These bootstrap samples were used in various ways.

We started by using the bootstrap samples as confidence bands on each of the edges. The more often an edge was found in the bootstrap samples, the more confident we were that the given edge is part of the DAG from which the data has been sampled from. We then defined a threshold t and removed all edges were found in less than a t fraction of all bootstrap samples. All other edges were kept. This gave us a summary graph that comprises all edges that had enough evidence to exist according to the bootstrap samples. We then either trimmed $\mathcal{S}^0$ or trimmed the summary graph $\mathcal{S}$ in order to remove all cycles. The downside of this approach is that data sets with underfitted DAGs, underfit even stronger after applying any of those methods. Moreover data sets with underfitted DAGs, underfit more, the stronger the underlying data is correlated. So as soon as the edge density came close to 0.3, this method worked quite badly. So we needed a different approach.

The following approach was prediction-based. We introduced a prediction algorithm for binary random variables. This prediction algorithm was then used to improve the predictive accuracy of the initial fitted network $\mathcal{S}^0$. This was done by estimating the Markov blanket for each vertex in the network. We then added iteratively new edges to the network until the predictive performance even decreased. The advantage of this approach is that the predictive performance decreases if our model underfits or if it overfits. So this algorithm adds edges to the model that underfit while being able to give the user some indication that the model starts overfitting since the predictive performance starts decreasing.

Next, we want to mention how this work could be continued.

Let us start with the bootstrap algorithm. Bootstrapping is the main algorithm used to reduce overfitting, and it is computationally very demanding, especially for large DAGs. This algorithm is parallelizable since iteration $i + 1$ does not depend on iteration i.

Additionally, a non-parametric bootstrap approach might be interesting to implement. This approach would take much less time and is much easier to code. The idea of non-parametric bootstrapping is to sample n data points from our data set with replacement. Then those data sets are used to refit the model. It might be interesting to compare the methods investigated during this work to the DAGs that a non-parametric bootstrapping algorithm returns.

Next, we would like to point out possible improvements in the trimming algorithms. The first trimming algorithm removes edges such that the resulting DAG has as many edges as possible. The probability that a given edge might be beneficial to keep, according to our data set, is not taken into account. The second algorithm takes that into account, without any success, since both trimming algorithms seem to work similarly well. We believe that there is much room for improvement in Algorithm 4.4. One approach is implementing a brute force algorithm that removes i different edges in iteration i. This process is repeated until we find a graph without any cycle for the first time. This approach, however, is most likely computationally unfeasible.

A different approach is to learn an order $(i_1, \ldots, i_d)$ in the set of vertices V. This means that if $i_m \prec i_n$, then whenever there exists an edge connecting v_{i_m} with v_{i_n}, it will be an edge going from vertex v_{i_m} to vertex v_{i_n}. If repeating this process for each edge in the graph, all cycles will be removed. Learning the order could be done using the bootstrap samples or using `searchHillClimber()` algorithm with multiple restarts. Let $\{\mathcal{S}^i\}_{i=1}^N$ be the set of DAGs from the bootstrap algorithm or the `searchHillClimber()`. Then for each $\mathcal{S} \in \{\mathcal{S}^i\}_{i=1}^N$ one can compute the score and the corresponding ordering. In the end, for each order, we end up with a score. The score of an ordering is defined to be the highest score of all scores, corresponding to the given order. The final order is the order with the highest score. One issue that one might run into is that orderings are not necessarily unique for the vertices whose positions are not uniquely identified. The true dependencies of the data might be represented by 2 disjoint DAGs. In this case $(v_{11}, \ldots, v_{1n_1}, v_{21}, \ldots, v_{2n_2})$ and $(v_{21}, \ldots, v_{2n_2}, v_{11}, \ldots, v_{1n_1})$ are two equivalent orders. However, if the dependencies of the data are best described via one graph that does not consist of multiple disjoint graphs, we need to find a unique ordering.

Alternatively, we could completely change the framework. So far, we have lived in the space of DAGs. Let us now live in the space of orderings $\prec$. We can then implement a `searchHillClimber()` algorithm in the space of orderings. We start with some random ordering. We compute the score of all possible DAGs with the given order $\prec$. Then the score of this particular ordering could be the maximum score, the mean score, or the mean score of some quantile of all scores corresponding to $\prec$. Then we apply some swap operator g on $\prec$. If $\prec = (X_{i_1}, \ldots, X_{i_j}, X_{i_{j+1}}, \ldots X_{i_d})$, then $g(\prec) = (X_{i_1}, \ldots, X_{i_{j+1}}, X_{i_j}, \ldots X_{i_d})$ for some random j. If the score of this new ordering increases, we define $\prec = g(\prec)$. Else, we apply g to $\prec$ with some different j. We can restart this process multiple times with different initial orderings. The highest scoring order is defined to be the order of the network. This process has been further discussed in Teyssier and Koller (2012).

The prediction algorithm has room for improvement as well. Remember that when predicting the value of node Y, the algorithm counts the number of occurrences for the given Markov blanket state and counts how often $Y = 0$ and how often $Y = 1$. Then a majority vote is conducted. However, if no observations of this form are apparent in the training set, the algorithm first removes exactly one vertex at a time from the Markov blanket. In the second iterations step, all pairs of vertices are removed, and it keeps going until a realization has been found. In which order the vertices are removed is so far random. We can do better by eliminating those vertices first, that if released, have the lowest decrease in the marginal likelihood score of the network. This might boost the speed of the prediction algorithm since it is more likely to find an observation in our data set if the score of the corresponding network is high. Moreover, the prediction accuracy itself might increase.

Additionally, the prediction algorithm could be generalized such that we can not only predict Binomial but Gaussian and Poisson variables as well.

Up to this point, we used all bootstrap samples to summarize them into one single DAG. It might be beneficial thinking of the distribution of our data to be a linear combination of weighted

DAGs instead of one single DAG. Therefore given a collection of DAGs $\{\mathcal{S}^i\}_{i=1}^N$, one needs to find weights $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_N)$, such that the distribution of $\boldsymbol{X} = (X_1, \ldots, X_d)$ is given by

$$P(\boldsymbol{X}) = \sum_{i=1}^{N} w_i \mathcal{S}^i$$

such that $(\omega_1, \ldots, \omega_N)$ is sparse, meaning that it has only a small number of non-zeros. The hard part is to learn ω. Moreover, the interpretation of a sum of weighted DAGs is no longer straightforward. The upside of such a method is the increased predictive accuracy. One can see a similarity to random trees. This framework considers a DAG as a tree and a bootstrap sample as a random forest.

In Chapter 4, we introduced two algorithms to decrease the level of underfitting in the initial structure $\mathcal{S}^0$. In Section 4.6 we have seen that in all cases, even after applying the algorithms introduced in Section 4.2 and Section 4.4 there were still random variables that seemed to be independent of all other random variables. However, the true distribution was a weakly connected graph. Therefore it might make sense to adjust the Algorithms introduced in Section 4.2 and Section 4.4. So far, the edge was added to the network, which had increased the predictive accuracy the most. But one could add a penalty term if the edge points to a vertex that already has a lot of parents. In those cases, vertices that are completely disconnected from the graph have a higher probability of being included in the graph.

Appendix A

R Code

In this section we list some R codes which are used for simulations or other purposes.

R-Code A.1. Simulation study setup for comparison of trimming algorithms from Chapter 3.2.

```
library(igraph)
source('my_bootstrap/trim.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')

whichCycle <- function(dag, d) {
  out <- list()
  for (i in 1:d) {
    print(paste0('i: ', i))
    for (j in 1:d) {
      if (i != j) {
        if (dag[i,j] == 0) {
          dag[i,j] <- 1
          if (length(return_cycles(dag)) > 0) {
            out[[length(out)+1]] <- c(i,j)
          }
          dag[i,j] <- 0
        }
      }
    }
  }
  return(out)
}

densities <- c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6)
n.datapoints <- c(300, 1200, 2700, 4800, 7500, 10800)
out <- list()
```

```
for (k in 1:6) {
  p <- 40
  cnt.greedy <- rep(0, p)
  cnt.score <- rep(0, p)
  equal <- rep(0, p)
  len <- c()
  for (i in 1:p) {
    set.seed(i)
    density <- densities[k]
    n.data <- n.datapoints[k]
    true.dag <- simulateDag(node.name = names, data.dists = dists,
                            edge.density = densities[k])
    if (sum(true.dag$dag) == 0) {
      next
    }
    both.observed <- F
    while (!both.observed) {
      s <- sample(seq(from = 1, to = 1000000), 1)
      data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                          n.chains = 1, n.adapt = 1000, n.thin = 10,
                          n.iter = 10*n.datapoints[k], verbose = F, seed = s)
      t <- apply(data, 1, as.numeric)
      if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
        both.observed <- T
      }
    }
    mycache <- buildScoreCache(data.df = data, data.dists = dists,
                               method = "bayes", max.parents = 8, verbose = F)
    mp.dag <- mostProbable(mycache, verbose = F)
    loops <- whichCycle(true.dag$dag, 10)
    if (length(loops)>0) {
      len <- c(len, length(loops))
      for (j in 1:length(loops)) {
        toTrim <- mp.dag$dag
        element <- loops[[j]]
        toTrim[element[1], element[2]] <- 1
        boot <- list()
        boot[[1]] <- toTrim
        greedy <- trim.G(boot, 0.5, method = 'greedy', mycache = NULL)[[1]]
        score.based <- trim.G(boot, 0.5, "score.based", mycache)[[1]]
        cnt.greedy[i] <- cnt.greedy[i] +
          compareDag(true.dag$dag, greedy)$`Hamming-distance`
        cnt.score[i] <- cnt.score[i] +
          compareDag(true.dag$dag, score.based)$`Hamming-distance`
        if (identical(score.based, greedy)) {
          equal[i] <- equal[i] + 1
        }
      }
    }
```

```
  }
  out[[k]] <- list()
  out[[k]][[1]] <- cnt.greedy
  out[[k]][[2]] <- cnt.score
  out[[k]][[3]] <- equal
  out[[k]][[4]] <- len
  out[[k]][[5]] <- mp.dag
  out[[k]][[6]] <- true.dag
  out[[k]][[7]] <- data
}

cnt.greedy
cnt.score
greedy <- rep(0, 6)
score <- rep(0, 6)
equal <- rep(0, 6)
for (j in 1:6) {
  for (i in 1:p) {
    greedy[j] <- greedy[j] + out[[j]][[1]][i]/out[[j]][[4]][i]
    score[j] <- score[j] + out[[j]][[2]][i]/out[[j]][[4]][i]
    equal[j] <- equal[j] + out[[j]][[3]][i]/out[[j]][[4]][i]
  }
}
greedy <- greedy/p
score <- score/p
equal <- equal/p
```

R-Code A.2. Setup for simulation study of Chapter 3.4 for edge density 0.1 and score function mlik.

```
library(abn)
library(igraph)
source('my_bootstrap/bootstrap.R')
source('my_bootstrap/trim.R')
source('my_bootstrap/prediction_based_mp_improvement.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')


# score = mlik, density = 0.1, n.datapoints = 50, R = 500
seeds <- c(1, 235, 400,  15, 293)
R <- 500
out <- list()
for (i in 1:5) {
  out[[i]] <- list()
  set.seed(seeds[i])
  density <- 0.1
  n.data <- 50
  true.dag <- simulateDag(node.name = names, data.dists = dists,
                          edge.density = density)
  both.observed <- F
  while (!both.observed) {
    s <- sample(seq(from = 1, to = 1000000), 1)
    data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                        n.chains = 1, n.adapt = 1000, n.thin = 10,
                        n.iter = 10*n.data, verbose = F, seed = s)
    t <- apply(data, 1, as.numeric)
    if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
      both.observed <- T
    }
  }
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                           n.chains = 1, n.adapt = 1000, n.thin = 10,
                           n.iter = 10*n.data, verbose = F,
                           seed = s+1)
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             method = "bayes", max.parents = 8, verbose = F)
  mp.dag <- mostProbable(mycache, verbose = F, score = 'mlik')
  t <- confusionMatrix(true.dag$dag, mp.dag$dag, dists)

  boot <- bootstrap(num.of.simulations = R,
                    data = data,
                    method = 'bayes',
```

```
                    score = 'mlik',
                    dists = dists,
                    max.par = 8,
                    ban = NULL,
                    retain = NULL,
                    n.chains = 1,
                    n.iter = n.data*10,
                    n.adapt = 1000,
                    n.thin = 10,
                    mode = 'dag',
                    path = NULL,
                    abn.object = NULL,
                    bug.file = NULL,
                    post.params = NULL,
                    return.datasets = FALSE,
                    keep.files = FALSE)

  thresholds <- c(0.2, 0.5, 0.7)
  obj.S <- trim.S(boot = boot, initial = mp.dag$dag, thresholds = thresholds)
  obj.G <- trim.G(boot = boot, thresholds = thresholds,
                  method = 'greedy', mycache = NULL)
  obj.G.2 <- trim.G(boot = boot, thresholds = thresholds,
                    method = 'score.based', mycache = mycache)

  cm.S <- list()
  cm.G <- list()
  cm.G.2<- list()
  for (j in 1:3) {
    cm.S[[j]] <- confusionMatrix(true.dag$dag, obj.S[[j]], dists)
    cm.G[[j]] <- confusionMatrix(true.dag$dag, obj.G[[j]], dists)
    cm.G.2[[j]] <- confusionMatrix(true.dag$dag, obj.G.2[[j]], dists)
  }

  out[[i]][[1]] <- true.dag
  out[[i]][[2]] <- mp.dag
  out[[i]][[3]] <- data
  out[[i]][[4]] <- data.test
  out[[i]][[5]] <- seeds[i]
  out[[i]][[6]] <- boot
  out[[i]][[7]] <- obj.S
  out[[i]][[8]] <- obj.G
  out[[i]][[9]] <- t
  out[[i]][[10]] <- cm.S
  out[[i]][[11]] <- cm.G
  out[[i]][[12]] <- obj.G.2
  out[[i]][[13]] <- cm.G.2
}
```

R-Code A.3. Setup for simulation study of Chapter 3.4 for edge density 0.2 and score function mlik.

```
library(abn)
library(igraph)
source('my_bootstrap/bootstrap.R')
source('my_bootstrap/trim.R')
source('my_bootstrap/prediction_based_mp_improvement.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')

seeds <- c(1, 2, 137, 316, 216)
R <- 500
out <- list()
for (i in 1:5) {
  out[[i]] <- list()
  set.seed(seeds[i])
  density <- 0.2
  n.data <- 75
  true.dag <- simulateDag(node.name = names, data.dists = dists,
                          edge.density = density)
  both.observed <- F
  while (!both.observed) {
    s <- sample(seq(from = 1, to = 1000000), 1)
    print(s)
    data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                        n.chains = 1, n.adapt = 1000, n.thin = 10,
                        n.iter = 10*n.data, verbose = F, seed = s)
    t <- apply(data, 1, as.numeric)
    if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
      both.observed <- T
    }
  }
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                           n.chains = 1, n.adapt = 1000, n.thin = 10,
                           n.iter = 10*n.data, verbose = F,
                           seed = s+1)
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             method = "bayes", max.parents = 8, verbose = F)
  mp.dag <- mostProbable(mycache, verbose = F, score = 'mlik')
  t <- confusionMatrix(true.dag$dag, mp.dag$dag, dists)

  boot <- bootstrap(num.of.simulations = R,
                    data = data,
                    method = 'bayes',
                    score = 'mlik',
```

```
                    dists = dists,
                    max.par = 8,
                    ban = NULL,
                    retain = NULL,
                    n.chains = 1,
                    n.iter = n.data*10,
                    n.adapt = 1000,
                    n.thin = 10,
                    mode = 'dag',
                    path = NULL,
                    abn.object = NULL,
                    bug.file = NULL,
                    post.params = NULL,
                    return.datasets = FALSE,
                    keep.files = FALSE)

  thresholds <- c(0.2, 0.5, 0.7)
  obj.S <- trim.S(boot = boot, initial = mp.dag$dag, thresholds = thresholds)
  obj.G <- trim.G(boot = boot, thresholds = thresholds,
                  method = 'greedy', mycache = NULL)
  obj.G.2 <- trim.G(boot = boot, thresholds = thresholds,
                    method = 'score.based', mycache = mycache)

  cm.S <- list()
  cm.G <- list()
  cm.G.2<- list()
  for (j in 1:3) {
    cm.S[[j]] <- confusionMatrix(true.dag$dag, obj.S[[j]], dists)
    cm.G[[j]] <- confusionMatrix(true.dag$dag, obj.G[[j]], dists)
    cm.G.2[[j]] <- confusionMatrix(true.dag$dag, obj.G.2[[j]], dists)
  }

  out[[i]][[1]] <- true.dag
  out[[i]][[2]] <- mp.dag
  out[[i]][[3]] <- data
  out[[i]][[4]] <- data.test
  out[[i]][[5]] <- seeds[i]
  out[[i]][[6]] <- boot
  out[[i]][[7]] <- obj.S
  out[[i]][[8]] <- obj.G
  out[[i]][[9]] <- t
  out[[i]][[10]] <- cm.S
  out[[i]][[11]] <- cm.G
  out[[i]][[12]] <- obj.G.2
  out[[i]][[13]] <- cm.G.2
}
```

R-Code A.4. Setup for simulation study of Chapter 3.4 for edge density 0.2 and score function BIC.

```
library(abn)
library(igraph)
source('my_bootstrap/bootstrap.R')
source('my_bootstrap/trim.R')
source('my_bootstrap/prediction_based_mp_improvement.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')
seeds <- c(1, 137, 316)
R <- 500
out <- list()
for (i in 1:3) {
  out[[i]] <- list()
  set.seed(seeds[i])
  density <- 0.2
  n.data <- 75
  true.dag <- simulateDag(node.name = names, data.dists = dists,
                          edge.density = density)
  both.observed <- F
  while (!both.observed) {
    s <- sample(seq(from = 1, to = 1000000), 1)
    data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                        n.chains = 1, n.adapt = 1000, n.thin = 10,
                        n.iter = 10*n.data, verbose = F,
                        seed = s)
    t <- apply(data, 1, as.numeric)
    if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
      both.observed <- T
    }
  }
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                           n.chains = 1, n.adapt = 1000, n.thin = 10,
                           n.iter = 10*n.data, verbose = F,
                           seed = s+1)
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             method = "mle", max.parents = 8, verbose = F)
  mp.dag <- mostProbable(mycache, verbose = F, score = 'bic')
  t <- confusionMatrix(true.dag$dag, mp.dag$dag, dists)

  boot <- bootstrap(num.of.simulations = R,
                    data = data,
                    method = 'mle',
                    score = 'bic',
                    dists = dists,
```

```
                    max.par = 8,
                    ban = NULL,
                    retain = NULL,
                    n.chains = 1,
                    n.iter = n.data*10,
                    n.adapt = 1000,
                    n.thin = 10,
                    mode = 'dag',
                    path = NULL,
                    abn.object = NULL,
                    bug.file = NULL,
                    post.params = NULL,
                    return.datasets = FALSE,
                    keep.files = FALSE)

  thresholds <- c(0.2, 0.5, 0.7)
  obj.S <- trim.S(boot = boot, initial = mp.dag$dag, thresholds = thresholds)
  obj.G <- trim.G(boot = boot, thresholds = thresholds,
                  method = 'greedy', mycache = NULL)
  obj.G.2 <- trim.G(boot = boot, thresholds = thresholds,
                    method = 'score.based', mycache = mycache)

  cm.S <- list()
  cm.G <- list()
  cm.G.2<- list()
  for (j in 1:3) {
    cm.S[[j]] <- confusionMatrix(true.dag$dag, obj.S[[j]], dists)
    cm.G[[j]] <- confusionMatrix(true.dag$dag, obj.G[[j]], dists)
    cm.G.2[[j]] <- confusionMatrix(true.dag$dag, obj.G.2[[j]], dists)
  }

  out[[i]][[1]] <- true.dag
  out[[i]][[2]] <- mp.dag
  out[[i]][[3]] <- data
  out[[i]][[4]] <- data.test
  out[[i]][[5]] <- seeds[i]
  out[[i]][[6]] <- boot
  out[[i]][[7]] <- obj.S
  out[[i]][[8]] <- obj.G
  out[[i]][[9]] <- t
  out[[i]][[10]] <- cm.S
  out[[i]][[11]] <- cm.G
  out[[i]][[12]] <- obj.G.2
  out[[i]][[13]] <- cm.G.2
}
```

R-Code A.5. Setup for simulation study of Chapter 3.4 for comparison of FP-FN when using mlik, BIC or AIC as score function.

```
library(abn)
source('my_bootstrap/prediction_based_mp_improvement.R')

mlik <- list()
bic <- list()
aic <- list()

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')

for (i in 1:400) {
  set.seed(i)
  density <- 0.3
  n.data <- 125
  true.dag <- simulateDag(node.name = names, data.dists = dists,
                          edge.density = density)
  both.observed <- F
  while (!both.observed) {
    s <- sample(seq(from = 1, to = 1000000), 1)
    data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                        n.chains = 1, n.adapt = 1000, n.thin = 10,
                        n.iter = 10*n.data, verbose = F, seed = s)
    t <- apply(data, 1, as.numeric)
    if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
      both.observed <- T
    }
  }
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                           n.chains = 1, n.adapt = 1000, n.thin = 10,
                           n.iter = 10*n.data, verbose = F,
                           seed = s+1)
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             method = "bayes", max.parents = 8, verbose = F)
  mp.dag <- mostProbable(mycache, verbose = F, score = 'mlik')
  mycache.bic <- buildScoreCache(data.df = data, data.dists = dists,
                                 method = "mle", max.parents = 8, verbose = F)
  mp.dag.bic <- mostProbable(mycache.bic, verbose = F, score = 'BIC')
  mp.dag.aic <- mostProbable(mycache, verbose = F, score = 'AIC')
  t <- confusionMatrix(true.dag$dag, mp.dag$dag, dists)
  t.bic <- confusionMatrix(true.dag$dag, mp.dag.bic$dag, dists)
  t.aic <- confusionMatrix(true.dag$dag, mp.dag.aic$dag, dists)
  mlik[[i]] <- t
  bic[[i]] <- t.bic
  aic[[i]] <- t.aic
```

```
}
```

R-Code A.6. Setup for simulation study of Chapter 3.4 for edge density 0.3 and score function BIC.

```
library(abn)
library(igraph)
source('my_bootstrap/bootstrap.R')
source('my_bootstrap/trim.R')
source('my_bootstrap/prediction_based_mp_improvement.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')
seeds <- c(151, 163, 232, 317, 364)
R <- 500
out <- list()
for (i in 1:5) {
  out[[i]] <- list()
  set.seed(seeds[i])
  density <- 0.3
  n.data <- 125
  true.dag <- simulateDag(node.name = names, data.dists = dists,
                          edge.density = density)
  both.observed <- F
  while (!both.observed) {
    s <- sample(seq(from = 1, to = 1000000), 1)
    data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                        n.chains = 1, n.adapt = 1000, n.thin = 10,
                        n.iter = 10*n.data, verbose = F,
                        seed = s)
    t <- apply(data, 1, as.numeric)
    if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
      both.observed <- T
    }
  }
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                           n.chains = 1, n.adapt = 1000, n.thin = 10,
                           n.iter = 10*n.data, verbose = F,
                           seed = s+1)
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             method = "mle", max.parents = 8, verbose = F)
  mp.dag <- mostProbable(mycache, verbose = F, score = 'bic')
  t <- confusionMatrix(true.dag$dag, mp.dag$dag, dists)

  boot <- bootstrap(num.of.simulations = R,
                    data = data,
                    method = 'mle',
                    score = 'bic',
                    dists = dists,
```

```
                    max.par = 8,
                    ban = NULL,
                    retain = NULL,
                    n.chains = 1,
                    n.iter = n.data*10,
                    n.adapt = 1000,
                    n.thin = 10,
                    mode = 'dag',
                    path = NULL,
                    abn.object = NULL,
                    bug.file = NULL,
                    post.params = NULL,
                    return.datasets = FALSE,
                    keep.files = FALSE)

  thresholds <- c(0.2, 0.5, 0.7)
  obj.S <- trim.S(boot = boot, initial = mp.dag$dag, thresholds = thresholds)
  obj.G <- trim.G(boot = boot, thresholds = thresholds,
                  method = 'greedy', mycache = NULL)
  obj.G.2 <- trim.G(boot = boot, thresholds = thresholds,
                    method = 'score.based', mycache = mycache)

  cm.S <- list()
  cm.G <- list()
  cm.G.2<- list()
  for (j in 1:3) {
    cm.S[[j]] <- confusionMatrix(true.dag$dag, obj.S[[j]], dists)
    cm.G[[j]] <- confusionMatrix(true.dag$dag, obj.G[[j]], dists)
    cm.G.2[[j]] <- confusionMatrix(true.dag$dag, obj.G.2[[j]], dists)
  }

  out[[i]][[1]] <- true.dag
  out[[i]][[2]] <- mp.dag
  out[[i]][[3]] <- data
  out[[i]][[4]] <- data.test
  out[[i]][[5]] <- seeds[i]
  out[[i]][[6]] <- boot
  out[[i]][[7]] <- obj.S
  out[[i]][[8]] <- obj.G
  out[[i]][[9]] <- t
  out[[i]][[10]] <- cm.S
  out[[i]][[11]] <- cm.G
  out[[i]][[12]] <- obj.G.2
  out[[i]][[13]] <- cm.G.2
}
```

R-Code A.7. Setup for simulation study of Chapter 3.4 for edge density 0.3 and score function AIC.

```
library(abn)
library(igraph)
source('my_bootstrap/bootstrap.R')
source('my_bootstrap/trim.R')
source('my_bootstrap/prediction_based_mp_improvement.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')
seeds <- c(151, 317, 364)
R <- 1
out <- list()
for (i in 1:3) {
  out[[i]] <- list()
  set.seed(seeds[i])
  density <- 0.3
  n.data <- 125
  true.dag <- simulateDag(node.name = names, data.dists = dists,
                          edge.density = density)
  both.observed <- F
  while (!both.observed) {
    s <- sample(seq(from = 1, to = 1000000), 1)
    data <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                        n.chains = 1, n.adapt = 1000, n.thin = 10,
                        n.iter = 10*n.data, verbose = F, seed = s)
    t <- apply(data, 1, as.numeric)
    if (max(apply(t, 1, sum)) != n.data & min(apply(t, 1, sum)) != 0) {
      both.observed <- T
    }
  }
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag$dag,
                           n.chains = 1, n.adapt = 1000, n.thin = 10,
                           n.iter = 10*n.data, verbose = F, seed = s+1)
  mycache <- buildScoreCache(data.df = data, data.dists = dists,
                             method = "mle", max.parents = 8, verbose = F)
  mp.dag <- mostProbable(mycache, verbose = F, score = 'aic')
  t <- confusionMatrix(true.dag$dag, mp.dag$dag, dists)

  boot <- bootstrap(num.of.simulations = R,
                    data = data,
                    method = 'mle',
                    score = 'aic',
                    dists = dists,
                    max.par = 8,
                    ban = NULL,
```

```
                    retain = NULL,
                    n.chains = 1,
                    n.iter = n.data*10,
                    n.adapt = 1000,
                    n.thin = 10,
                    mode = 'dag',
                    path = NULL,
                    abn.object = NULL,
                    bug.file = NULL,
                    post.params = NULL,
                    return.datasets = FALSE,
                    keep.files = FALSE)

  thresholds <- c(0.2, 0.5, 0.7)
  obj.S <- trim.S(boot = boot, initial = mp.dag$dag, thresholds = thresholds)
  obj.G <- trim.G(boot = boot, thresholds = thresholds,
                  method = 'greedy', mycache = NULL)
  obj.G.2 <- trim.G(boot = boot, thresholds = thresholds,
                    method = 'score.based', mycache = mycache)

  cm.S <- list()
  cm.G <- list()
  cm.G.2<- list()
  for (j in 1:3) {
    cm.S[[j]] <- confusionMatrix(true.dag$dag, obj.S[[j]], dists)
    cm.G[[j]] <- confusionMatrix(true.dag$dag, obj.G[[j]], dists)
    cm.G.2[[j]] <- confusionMatrix(true.dag$dag, obj.G.2[[j]], dists)
  }

  out[[i]][[1]] <- true.dag
  out[[i]][[2]] <- mp.dag
  out[[i]][[3]] <- data
  out[[i]][[4]] <- data.test
  out[[i]][[5]] <- seeds[i]
  out[[i]][[6]] <- boot
  out[[i]][[7]] <- obj.S
  out[[i]][[8]] <- obj.G
  out[[i]][[9]] <- t
  out[[i]][[10]] <- cm.S
  out[[i]][[11]] <- cm.G
  out[[i]][[12]] <- obj.G.2
  out[[i]][[13]] <- cm.G.2
}
```

R-Code A.8. Setup for simulation study of Chapter 4.6 to compare the two algorithms introduced in Chapters 4.2 and 4.4.

```
library(abn)
library(igraph)
library(Matrix)
source('my_bootstrap/bootstrap.R')
source('my_bootstrap/trim.R')
source('my_bootstrap/prediction_based_mp_improvement.R')
source('my_bootstrap/predict.R')

names <- c('x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9')
dists <- list(x0 = 'binomial', x1 = 'binomial', x2 = 'binomial',
              x3 = 'binomial', x4 = 'binomial', x5 = 'binomial',
              x6 = 'binomial', x7 = 'binomial', x8 = 'binomial',
              x9 = 'binomial')

# This first part is for Algorithm 1
density <- 0.3
n.data <- 100
seeds <- seq(from = 1, to = 5, by = 1)
out <- list()

for (i in 1:5) {
  out[[i]] <- list()
  set.seed(seeds[i])
  true.dag <- simulateDag(names, dists, density)
  true.dag <- true.dag$dag

  data.train <- simulateAbn(data.dists = dists, data.param = true.dag,
                            n.adapt = 1000, n.chains = 1, n.thin = 10,
                            n.iter = 10*n.data, seed = seeds[i])
  data.test <- simulateAbn(data.dists = dists, data.param = true.dag,
                           n.adapt = 1000, n.chains = 1, n.thin = 10,
                           n.iter = 10*n.data, seed = seeds[i]+1000)

  mycache <- buildScoreCache(data.df = data.train, data.dists = dists,
                             method = 'bayes', max.parents = 6)
  mp.dag <- mostProbable(score.cache = mycache)
  mp.dag <- mp.dag$dag

  boot <- bootstrap(num.of.simulations = 1000, data = data.train,
                    method = 'bayes', score = 'mlik', dists = dists,
                    max.par = 8, ban = NULL, retain = NULL, n.chains = 1,
                    n.iter = 1000, n.adapt = 1000, n.thin = 10)

  accuracies <- constructAccuracyMatrix(data.train = data.train,
                                        data.test = data.test,
                                        boot = boot, dists = dists)
```

```
  addedEdges1 <- addEdges(mp.dag, names, accuracies,
                          boot, data.train, data.test)

  out[[i]][[1]] <- true.dag
  out[[i]][[2]] <- mp.dag
  out[[i]][[3]] <- data.train
  out[[i]][[4]] <- data.test
  out[[i]][[5]] <- boot
  out[[i]][[6]] <- accuracies
  out[[i]][[7]] <- addedEdges1
  out[[i]][[8]] <- seeds[i]
}

# Approach 1 for all
for (i in 1:5) {
  plot(rowSums(out[[i]][[7]][[1]]))
}

# This second part is for algorithm 2
# Approach 2 for Simulations 1
plotAccuracies(out[[1]][[6]])
acc.threshold <- c(0.66, 0.95, 0.75, 0.58, 0.58, 0.78, 0.81, 0.55, 0.55, 0.65)
count <- countMb(out[[1]][[6]], acc.threshold, out[[1]][[5]], dists)
count

num <- c(3, 1, 3, 2, 2, 2, 2, 1, 2, 1)
markov.blanket <- returnMb(count, num)
addedEdges <- addEdges(out[[1]][[2]], names, out[[1]][[6]], out[[1]][[5]],
                       out[[1]][[3]], out[[1]][[4]], 10, markov.blanket)
acc <- apply(addedEdges[[1]], 1, sum)
plot(acc)

compareDag(out[[1]][[1]], out[[1]][[2]])$`Hamming-distance`
compareDag(out[[1]][[1]], addedEdges[[3]][[4]])$`Hamming-distance`
plotAbn(out[[1]][[1]], dists)
plotAbn(out[[1]][[2]], dists)
plotAbn(addedEdges[[3]][[4]], dists)

out[[1]][[9]] <- acc.threshold
out[[1]][[10]] <- num
out[[1]][[11]] <- addedEdges
out[[1]][[12]] <- addedEdges[[3]][[4]]


# Approach 2 for Simulations 2
plotAccuracies(out[[2]][[6]])
acc.threshold <- c(0.72, 0.58, 0.64, 0.97, 0.58, 0.45, 0.82, 0.55, 0.48, 0.68)
count <- countMb(out[[2]][[6]], acc.threshold, out[[2]][[5]], dists)
count
```

```
num <- c(1, 2, 1, 5, 2, 2, 2, 2, 1, 2)
markov.blanket <- returnMb(count, num)
addedEdges <- addEdges(out[[2]][[2]], names, out[[2]][[6]], out[[2]][[5]],
                       out[[2]][[3]], out[[2]][[4]], 10, markov.blanket)
acc <- apply(addedEdges[[1]], 1, sum)
plot(acc)

compareDag(out[[2]][[1]], out[[2]][[2]])$`Hamming-distance`
compareDag(out[[2]][[1]], addedEdges[[3]][[5]])$`Hamming-distance`
plotAbn(out[[2]][[1]], dists)
plotAbn(out[[2]][[2]], dists)
plotAbn(addedEdges[[3]][[5]], dists)

out[[2]][[9]] <- acc.threshold
out[[2]][[10]] <- num
out[[2]][[11]] <- addedEdges
out[[2]][[12]] <- addedEdges[[3]][[5]]


# Approach 2 for Simulations 3
plotAccuracies(out[[3]][[6]])
acc.threshold <- c(0.61, 0.56, 0.58, 0.57, 0.63, 0.58, 0.60, 0.70, 0.77, 0.63)
count <- countMb(out[[3]][[6]], acc.threshold, out[[3]][[5]], dists)
count

num <- c(2, 2, 3, 3, 3, 2, 2, 2, 4, 3)
markov.blanket <- returnMb(count, num)
addedEdges <- addEdges(out[[3]][[2]], names, out[[3]][[6]], out[[3]][[5]],
                       out[[3]][[3]], out[[3]][[4]], 10, markov.blanket)
acc <- apply(addedEdges[[1]], 1, sum)
plot(acc)

compareDag(out[[3]][[1]], out[[3]][[2]])$`Hamming-distance`
compareDag(out[[3]][[1]], addedEdges[[3]][[5]])$`Hamming-distance`
plotAbn(out[[3]][[1]], dists)
plotAbn(out[[3]][[2]], dists)
plotAbn(addedEdges[[3]][[5]], dists)

out[[3]][[9]] <- acc.threshold
out[[3]][[10]] <- num
out[[3]][[11]] <- addedEdges
out[[3]][[12]] <- addedEdges[[3]][[5]]


# Approach 2 for Simulations 4
plotAccuracies(out[[4]][[6]])
acc.threshold <- c(0.72, 0.70, 0.52, 0.80, 0.80, 0.56, 0.86, 0.87, 0.65, 0.84)
count <- countMb(out[[4]][[6]], acc.threshold, out[[4]][[5]], dists)
count
```

```
num <- c(4, 5, 3, 4, 3, 3, 4, 5, 4, 4)
markov.blanket <- returnMb(count, num)
addedEdges <- addEdges(out[[4]][[2]], names, out[[4]][[6]], out[[4]][[5]],
                       out[[4]][[3]], out[[4]][[4]], 10, markov.blanket)
acc <- apply(addedEdges[[1]], 1, sum)
plot(acc)

compareDag(out[[4]][[1]], out[[4]][[2]])$`Hamming-distance`
compareDag(out[[4]][[1]], addedEdges[[3]][[9]])$`Hamming-distance`
plotAbn(out[[4]][[1]], dists)
plotAbn(out[[4]][[2]], dists)
plotAbn(addedEdges[[3]][[9]], dists)

out[[4]][[9]] <- acc.threshold
out[[4]][[10]] <- num
out[[4]][[11]] <- addedEdges
out[[4]][[12]] <- addedEdges[[3]][[9]]


# Approach 2 for Simulations 5
plotAccuracies(out[[5]][[6]])
acc.threshold <- c(0.8, 0.75, 0.77, 0.85, 0.73, 0.60, 0.48, 0.70, 0.47, 0.58)
count <- countMb(out[[5]][[6]], acc.threshold, out[[5]][[5]], dists)
count

num <- c(4, 4, 4, 4, 4, 3, 3, 4, 4, 4)
markov.blanket <- returnMb(count, num)
addedEdges <- addEdges(out[[5]][[2]], names, out[[5]][[6]], out[[5]][[5]],
                       out[[5]][[3]], out[[5]][[4]], 10, markov.blanket)
acc <- apply(addedEdges[[1]], 1, sum)
plot(acc)

compareDag(out[[5]][[1]], out[[5]][[2]])$`Hamming-distance`
compareDag(out[[5]][[1]], addedEdges[[3]][[5]])$`Hamming-distance`
plotAbn(out[[5]][[1]], dists)
plotAbn(out[[5]][[2]], dists)
plotAbn(addedEdges[[3]][[5]], dists)

out[[5]][[9]] <- acc.threshold
out[[5]][[10]] <- num
out[[5]][[11]] <- addedEdges
out[[5]][[12]] <- addedEdges[[3]][[5]]
```

Bibliography

Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, Complex Systems 1695. v, 28

Friedman, N., Goldszmidt, M., and Wyner, A. J. (2013). Data analysis with bayesian networks: A bootstrap approach. *CoRR*, abs/1301.6695. v, 13

Held, L. and Bové, D. S. (2020). *Likelihood and Bayesian inference : with applications in biology and medicine*. Statistics for biology and health. Springer, Berlin. 4

Kratzer, G., Lewis, F. I., Comin, A., Pittavino, M., and Furrer, R. (2019a). Additive Bayesian network modelling with the R package abn. *arXiv preprint arXiv:1911.09006*. v, 13

Kratzer, G., Lewis, F. I., Comin, A., Pittavino, M., and Furrer, R. (2019b). Additive bayesian network modelling with the r package abn. 13

MacKay, D. J. C. (1992). Bayesian Interpolation. *Neural Computation*, 4 415–447. 3

Pittavino, M. (2016). Additive bayesian networks for multivariate data: parameter learning, model fitting and applications in veterinary epidemiology. 1, 13

Plummer, M. (2018). Bayesian graphical models using mcmc. v, 13

Plummer, M., Best, N., Cowles, K., and Vines, K. (2006). Coda: Convergence diagnosis and output analysis for mcmc. *R News*, 6 7–11. v, 13

Teyssier, M. and Koller, D. (2012). Ordering-based search: A simple and effective algorithm for learning bayesian networks. *CoRR*, abs/1207.1429. 62