Symbolic Parameter Estimation of Continuous-Time Markov Chains

Sebastian Aaholm[*], Lars Emanuel Hansen[†], Daniel Runge Petersen[‡],

Abstract—This is a placeholder abstract. The whole template is used in semester projects at AAU.

Index Terms—Formal Verification, Parameter Estimation, Decision Diagram

1 Introduction

Markov models are a class of probabilistic models that are used to describe the evolution of a system over time. A Markov model has the Markov property, which states that the future behavior of the system depends only on its current state and not on its past history [1]. This property simplifies analysis by focusing only on the present state, making Markov models especially useful for systems where memory-less behavior is a reasonable assumption.

Markov models are widely used in various fields, such as biology, finance, and computer science, to model systems that exhibit stochastic behavior [2–5]. As such, their analysis has a wide range of applications.

An example of a Markov model, is a simple weather model, if today is sunny, there might be an 80% chance of sun tomorrow and a 20% chance of rain. Similarly, if today is rainy, there might be a 70% chance of rain tomorrow and a 30% chance of sun.

Model checking is a technique used to verify the correctness of Markov models by comparing the predictions of the model with observed data. Model checking is widely used in the verification of Markov models, where the model is analyzed to check if it satisfies certain properties [6]. It ensures reliability and correctness in critical systems, from traffic controls to industrial automation and communication protocols [6]. It is also used to check if the model satisfies certain properties, such as reachability, can we reach a desired state and safety properties, can we avoid going a specific sequence of states.

A real world example of model checking is the verification of a traffic light system, where the model is analyzed to check if the traffic lights are working correctly. For reachability, we can ask: *can a traffic light system always cycle back to green after being red?*. For safety properties we can ask, *can a traffic light system avoid having both lights green at the same time?*.

There exists several tools for model checking, such as PRISM [7] and Storm [8], which are widely used in the verification of Markov models. These tools use symbolic representations to represent the model and perform the operations required for model checking. The limitation of these tools is that they do not support parameter estimation, which makes them unsuitable for learning the parameters of the model from data.

- *All authors are with the Dept. of Computer Science, Aalborg University, Aalborg, Denmark*
- *E-mails: [*]saahol20, [†]leha20, [‡]dpet20 @student.aau.dk*

Parameter estimation is a crucial step in the analysis of Markov models, as the analysis of the model depends on the accuracy of the estimated parameters, particularly when in a timing and probabilistic behaviour [9]. Parameter estimation is the process of estimating the parameters of the model from observed data, which is used to make predictions about the system's behavior.

These parameters are used to ensure that the model accurately represents the system's behavior and dynamics and to make accurate predictions about the system's future behavior. Accurate parameter estimation is essential for making reliable predictions and validating model behavior, with applications ranging from healthcare diagnostics to network security [9].

The Baum-Welch algorithm is a widely used method for estimating the parameters of Markov models [10]. The algorithm uses the Expectation-Maximization (EM) framework to iteratively update the parameters of the model until convergence [11]. The Baum-Welch algorithm is computationally expensive for large models, as it uses matrices to represent the model, which has a space complexity that grows quadratically with the number of states in the model. This makes the algorithm computationally expensive for large models, as the memory requirements grow rapidly with the size of the model [12].

Addressing these challenges requires innovative techniques, such as symbolic representations, which reduce memory consumption while preserving accuracy.

1.1 Related Works

PRISM [7] is a widely used probabilistic model checker designed to verify the correctness of Markov models. PRISM has developed a language for specifying models and properties, called the PRISM Language, which is widely used in the field of probabilistic model checking.

When models are specified in the PRISM Language, PRISM can provide a symbolic representation such as Algebraic Decision Diagrams (ADDs) to represent and manipulate the models efficiently, enabling the verification of properties like reachability and safety.

However, PRISM does not support parameter estimation, making it unsuitable for tasks requiring the inference of model parameters from observed data.

Storm [8] is another state-of-the-art probabilistic model checker that shares many similarities with PRISM. Like PRISM, it can provide a symbolic representations to handle large models efficiently and focuses on verifying properties of Markov models.

It has a parser to read models specified in the PRISM Language, making it easy to use for users familiar with PRISM. Storm has been optimized for scalability and flexibility, supporting a wide range of model types and verification tasks.

Additionally, Storm is open-source and has a large user base, making it a popular choice for probabilistic model checking. Despite these strengths, Storm also lacks support for parameter estimation, limiting its utility for learning model parameters from data.

Jajapy [13] is a Python-based tool designed for estimating parameters in parametric models using the Baum-Welch algorithm. It employs a matrix representation of the model and implements the necessary operations for parameter estimation through standard matrix computations without standard matrix libraries.

While accessible and straightforward, Jajapy is hindered by the space complexity inherent in its iterative-based calculation. This limitation makes it computationally expensive for large-scale models, as memory requirements grow quadratically with the number of states in the system.

SUDD [14] builds upon the limitations of Jajapy by introducing a symbolic representation for the forward-backward algorithm. Specifically, it leverages ADDs to reduce memory consumption and improve the runtime performance of the Baum-Welch algorithm. By employing ADD-based computations, SUDD provides a significant improvement in scalability, making it feasible to handle larger models.

However, the implementation is limited to a subset of the Baum-Welch algorithm, focusing primarily on forward-backward computations without addressing the full parameter estimation process.

Also the update step in the Baum-Welch algorithm requires SUDD with an explicit state space representation of the model, which limits the scalability of the algorithm.

Our work bridges the gap between parameter estimation tools (e.g. Jajapy and SUDD) and model checking tools (e.g. PRISM and Storm). We do this by extending Storm by integrating the Baum-Welch algorithm for with symbolic representations to improve the runtime performance of parameter estimation for Markov models primarily Hidden Markov Models (HMMs).

This integration of parameter learning with symbolic computation addresses a critical limitation in the current landscape of tools for Markov models. As a result, Storm can now be used to estimate parameters for HMMs from data, enabling the accurate modeling of complex systems.

2 Preliminaries

In this section, we introduce the necessary background concepts and definitions that are essential for understanding the subsequent sections. We begin by defining the key concepts of a HMM and then describe how a HMM can be represented using matrices. We then introduce the Baum-Welch algorithm, which is used to estimate the parameters of a HMM from observed data. We describe all the steps involved in the Baum-Welch algorithm, namely the forward-backward algorithm and the parameter's update. Finally, we discuss how the Baum-Welch algorithm can be implemented using matrix operations to efficiently compute the forward and backward variables, intermediate variables, and parameter updates.

2.1 Hidden Markov Models

HMMs were introduced by Baum and Petrie in 1966 [15]. HMM are a class of probabilistic models that are widely used to model sequences of observations dependent on some underlying hidden states. These models consist of two main components: observations and hidden states. The observations are the visible data emitted by the model, while the hidden states represent the underlying process that generates these observations. The objective of an HMM is to infer the hidden states based on the observations. HMMs have applications in fields such as speech recognition [16], bioinformatics [17], and natural language processing [18]. HMMs was chosen as the model of choice for this project due to its versatility and ability to model complex systems.

Definition 1 (Hidden Markov Model). A Hidden Markov Model (HMM) is a tuple $\mathcal{M} = (S, \mathcal{L}, \ell, \tau, \pi)$, where:

- S is a finite set of states.
- $\mathcal{L}$ is a finite set of labels.
- $\ell : S \to D(\mathcal{L})$ is the emission function.
- $\tau : S \to D(S)$ is the transition function.
- $\pi \in D(S)$ is the initial distribution.

Here, D denotes the set of discrete probability distributions over a finite set. The model emits a label l in state s with probability $\ell(s)(l)$, transitions between states with probability $\tau(s)(s')$, and starts in state s with probability $\pi(s)$.

2.2 Matrix Representation of HMMs

HMMs can be represented using various matrices. The emission function ℓ can be represented as a matrix $\boldsymbol{\omega}$ where $\boldsymbol{\omega}_{s,l} = \ell(s)(l)$. The matrix $\boldsymbol{\omega}$ has the size $|S| \times |\mathcal{L}|$. The sum of each row in the matrix $\boldsymbol{\omega}$ is equal to one, reflecting the total probability of emitting all labels from a given state.

$$\boldsymbol{\omega} = \begin{bmatrix} \ell(s_1)(l_1) & \cdots & \ell(s_1)(l_{|\mathcal{L}|}) \\ \vdots & \ddots & \vdots \\ \ell(s_{|S|})(l_1) & \cdots & \ell(s_{|S|})(l_{|\mathcal{L}|}) \end{bmatrix}$$

The transition function τ can be represented as a stochastic matrix $\boldsymbol{P}$ where $\boldsymbol{P}_{s,s'} = \tau(s)(s')$. The matrix $\boldsymbol{P}$ has the size $|S| \times |S|$. The sum of each row in $\boldsymbol{P}$ is equal to one, reflecting the total probability of transitioning from a given state to all other states.

$$\boldsymbol{P} = \begin{bmatrix} \tau(s_1)(s_1) & \cdots & \tau(s_1)(s_{|S|}) \\ \vdots & \ddots & \vdots \\ \tau(s_{|S|})(s_1) & \cdots & \tau(s_{|S|})(s_{|S|}) \end{bmatrix}$$

The initial distribution π can be represented as a vector $\boldsymbol{\pi}$ where $\boldsymbol{\pi}_s = \pi(s)$. The vector $\boldsymbol{\pi}$ has the size $|S|$. The sum of all elements in $\boldsymbol{\pi}$ is equal to one, reflecting the fact that $\pi \in D(s)$.

$$\boldsymbol{\pi} = \begin{bmatrix} \pi(s_1) \\ \vdots \\ \pi(s_{|S|}) \end{bmatrix}$$

2.3 Observations and Hidden States

An HMM operates on a multiset of observations, denoted as $\mathcal{O} = O_1, O_2, \ldots, O_N$, where each O_i is a observation sequence of labels $\mathbf{O} = o_1, o_2, \ldots, o_{|\mathbf{O}|-1}$.

This problem is commonly approximated using the Baum-Welch algorithm, a widely used method for estimating the probabilities of an HMM and finding the most likely hidden state sequence.

2.4 The Baum-Welch Algorithm

The Baum-Welch algorithm is a fundamental method for estimating the parameters of a HMM given a sequence of observations. These parameters include the emission matrix ω, the transition matrix P, and the initial state distribution π. The algorithm is widely recognized as the standard approach for training HMMs and was chosen for this project due to its ability to estimate these parameters without prior knowledge of the hidden states that generated the observations.

The Baum-Welch algorithm applies the Expectation-Maximization (EM) framework to iteratively improve the likelihood of the observed data under the current model parameters. It consists of the following steps:

1) **Initialization:** Begin with initial estimates for the HMM parameters (π, P, ω).
2) **Expectation Step (E-step):** Compute the expected counts of the latent variables, i.e., the hidden states, based on the observation sequence and the current model parameters. That is we compute the probabilities of observing the sequence up to time t, given that the HMM is in state s at time t and the probabilities of observing the sequence from time $t+1$ to the end, given that the HMM is in state s at time t.
3) **Maximization Step (M-step):** Update the HMM parameters (π, P, ω) to maximize the likelihood of the observed data based on the expected counts computed in the E-step.
4) **Iteration:** Repeat the E-step and M-step until convergence, i.e., when the change in likelihood between iterations falls below a predefined threshold.

The Baum Welch algorithm seeks to estimate the parameters τ, π, and ω of a HMM model $\mathcal{M}$ so that it maximises the likelihood function $l(\mathcal{M}|O)$. That is, the probability that the HMM $\mathcal{M}$ has independently generated each observation sequence $O_1, \ldots, O_N$.

Starting with an initial hypothesis $\mathbf{x}_0 = (\pi, P, \omega)$, the algorithm produces a sequence of parameter estimates $\mathbf{x}_1, \mathbf{x}_2, \ldots$, where each new estimate improves upon the previous one. The process terminates when the improvement in likelihood is sufficiently small, satisfying the convergence criterion:

$$||l(x_n,)|| < \epsilon$$

Where $l(x_n, o)$ is the likelihood of all the observations under the parameters x_n, and $\epsilon > 0$ is a small threshold.

The steps of the Baum-Welch algorithm are detailed in the following sections, including the initialization of the HMM parameters, the forward-backward algorithm, and the update algorithm, see subsection 2.5, 2.6, and 2.7 respectively.

2.5 Initialization of HMM Parameters

Before starting the Baum-Welch algorithm, we need to initialize the model parameters: the emission matrix ω, the transition matrix P, and the initial state distribution π.

Since the algorithm is designed to converge iteratively toward a locally optimal parameter set, the initial estimates do not need to be exact. However, reasonable initialization can accelerate convergence and improve numerical stability [19].

As we work with models that we have no prior knowledge of, meaning we dont know the number of observations, or what states that generated the observations, we cannot initialize th model parameters based on some domain knowledge. Therefore we need to initialize the parameters based on some strategy. A common approach to initialize these parameters is as one of the following strategies:

1) Random initialization:
   - Assign random probabilities to for the initial state distribution π_s, such that $\sum_{s\in S} \pi_s = 1$.
   - Assign random probabilities to each transition $P_{s,s'}$, such that $\sum_{s'\in S} P_{s,s'} = 1$.
   - Assign random probabilities to each emission $\omega_{s,l}$, such that $\sum_{l\in\mathcal{L}} \omega_{s,l} = 1$ for all $s \in S$.
2) Uniform initialization:
   - Set $\pi_s = \frac{1}{|S|}$ for all $s \in S$.
   - Set $P_{s,s'} = \frac{1}{|S|}$ for all $s, s' \in S$.
   - Set $\omega_{s,l} = \frac{1}{|\mathcal{L}|}$ for all $s \in S, l \in \mathcal{L}$.

We initialize the parameters using random initialization, as it provides a diverse set of initial values that can help avoid local optima. The uniform initialization is the worst choice as it provides a very limited set of initial values that can lead to local optima, and it is not recommended for practical use.

These initialization strategies provide a starting point for the Baum-Welch algorithm to iteratively refine the model parameters based on the observed data.

2.6 Forward-Backward Algorithm

For a given HMM $\mathcal{M}$, the forward-backward algorithm computes the forward and backward variables, $\alpha_s(t)$ and $\beta_s(t)$, for each observation sequence $o_0, o_1, \ldots, o_{|\mathbf{o}|-1} = \mathbf{o} \in \mathcal{O}$. The forward variable $\alpha_s(t)$ represents the likelihood of observing the partial sequence $o_0, o_1, \ldots, o_t$ and being in state s at time t, given the model $\mathcal{M}$. The backward variable $\beta_s(t)$ represents the likelihood of observing the partial sequence $o_{t+1}, o_{t+2}, \ldots, o_{|\mathbf{o}|-1}$ given state s at time t and the model $\mathcal{M}$.

The forward variable $\alpha_s(t)$ and backward variable $\beta_s(t)$ can be computed recursively as follows:

$$\alpha_s(t) = \begin{cases} \omega_{s,o_t}\pi_s & \text{if } t = 0 \\ \omega_{s,o_t} \sum_{s'\in S} P_{s's}\alpha_{s'}(t-1) & \text{if } 0 < t \leq |\mathbf{o}| - 1 \end{cases} \tag{1}$$

$$\beta_s(t) = \begin{cases} \mathbb{1} & \text{if } t = |\mathbf{o}| - 1 \\ \sum_{s'\in S} P_{ss'}\omega_{s'}(t+1)\beta_{s'}(t+1) & \text{if } 0 \leq t < |\mathbf{o}| - 1 \end{cases} \tag{2}$$

Here, ω_{s,o_t} is the likelihood of observing o_t given that the state at time t is s and the model $\mathcal{M}$, formally $\omega_{s,o_t} =$

$l(o_t \mid s, \mathcal{M}) = \ell(s)(o_t)$. Meaning that ω_{s,o_t} is the probability of observing the label o_t in state s.

The forward-backward algorithm computes the forward and backward variables for each state s and time t in the observation sequence $\mathbf{o}$, providing a comprehensive view of the likelihood of the observed data under the model.

In preparation for later discussions we would like to draw the attention to the fact that the above recurrences can be solved using dynamic programming requiring one to use $\Theta(|S| \times |(|\mathbf{o}| - 1)|)$ space.

2.7 Update Algorithm

The update algorithm refines the parameter values of the HMM model based on the observed data and the forward and backward variables computed in the forward-backward procedure. Given the forward and backward variables $\alpha_s(t)$ and $\beta_s(t)$, the update algorithm aims to maximize the likelihood of the observed data by adjusting the parameter values.

The update step is based on the expected sufficient statistics of the latent variables, which are the unobserved state sequences corresponding to the observations.

2.7.1 Intermediate Variables

We need to calculate the intermediate variables $\gamma_s(t)$ and $\xi_{ss'}(t)$. $\gamma_s(t)$ represent the expected number of times the model is in state s at time t given that the sequence O was observed and $\xi_{ss'}(t)$ represent the expected number of transitions from state s to state s' at time t given that the sequence O was observed.

For a given HMM $\mathcal{M}$, the intermediate variables, $\gamma_s(t)$ and $\xi_{ss'}(t)$, are computed for each observation sequence $o_0, o_1, \ldots, o_{|\mathbf{o}|-1} = \mathbf{o} \in \mathcal{O}$. These variables are computed as follows:

$$\gamma_s(t) = \frac{\alpha_s(t)\beta_s(t)}{\sum_{s' \in S} \alpha_{s'}(t)\beta_{s'}(t)} \tag{3}$$

In Equation 3, the numerator is the product of the forward variable $\alpha_s(t)$ and the backward variable $\beta_s(t)$, representing the joint probability of observing the entire sequence given that the model passed by state s at time t. The denominator represents the probability of the observation sequence.

$$\xi_{ss'}(t) = \frac{\alpha_s(t) P_{ss'} \omega_{s'}(t+1) \beta_{s'}(t+1)}{\sum_{s''} \alpha_{s''}(t)\beta_{s''}(t)} \tag{4}$$

In Equation 4, the numerator is the joint probability of observing the sequence given that the model transitions from state s to state s' at time t. The denominator represents the probability of the observation sequence.

The terms $\gamma_s(t)$ and $\xi_{ss'}(t)$ are normalized to ensure they represent probabilities. For $\gamma_s(t)$, this involves dividing by the total probability across all states at time t, while for $\xi_{ss'}(t)$, normalization occurs over all possible transitions at time t.

2.7.2 Parameter Update

The parameter update step refines the parameter values of the model based on the earlier computed intermediate variables $\gamma_s(t)$ and $\xi_{ss'}(t)$. The update algorithm aims to maximize the expected likelihood of the observed data under the model by adjusting the parameter values.

Once $\gamma_s(t)$ and $\xi_{ss'}(t)$ are computed for all states s, s' and all time steps t for every observation sequence, the model parameters can be updated to maximize the expected log-likelihood.

Transition Probabilities (P): We update the transition probabilities based on the expected number of transitions between states:

$$P_{ss'} = \frac{\sum_{t=1}^{|\mathbf{o}|-1} \xi_{ss'}(t)}{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)} \tag{5}$$

The numerator sums the expected number of transitions from state s to state s' over all time steps. The denominator sums the expected number of times the model is in state s over all time steps, ensuring $P_{ss'}$ is normalized across all s'.

Emission Probabilities (ω): We update the emission probabilities based on the expected occupancy of state s and the corresponding observations, meaning the probability of observing the specific label o in state s.

The update is given by:

$$\omega_{s,l} = \frac{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t) [\![o_t = l]\!]}{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)} \tag{6}$$

The numerator sums $\gamma_s(t)$ for all time steps t where the observed value $o_t = l.$, meaning the model is in state s and emits the observation l. The denominator sums $\gamma_s(t)$ for all time steps t where the model is in a given state s.

Initial Probabilities (π): We update the initial probabilities based on the expected occupancy of state s at $t = 1$:

$$\pi_s = \gamma_s(1) \tag{7}$$

We can then update the parameters $\mathbf{x}$ by maximizing the expected log-likelihood of the observed data under the model. The update algorithm iteratively refines the parameter values until convergence is reached.

2.8 Matrix Operations

The Baum-Welch algorithm can be implemented using matrix operations to efficiently compute the forward and backward variables, intermediate variables, and parameter updates.

Given a HMM $\mathcal{M}$ with parameters $\boldsymbol{\omega}$, $\boldsymbol{P}$, and $\boldsymbol{\pi}$, and an observation sequence $\mathbf{o}$, the forward and backward variables $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ can be computed using matrix operations as follows:

$$\boldsymbol{\alpha}_t = \begin{cases} \boldsymbol{\omega}_0 \circ \boldsymbol{\pi} & \text{if } t = 0 \\ \boldsymbol{\omega}_t \circ (\boldsymbol{P}^\top \boldsymbol{\alpha}_{t-1}) & \text{if } 0 < t \leq |\mathbf{o}| - 1 \end{cases} \tag{8}$$

$$\boldsymbol{\beta}_t = \begin{cases} \mathbb{1} & \text{if } t = |\mathbf{o}| - 1 \\ \boldsymbol{P}\,(\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1}) & \text{if } 0 \leq t < |\mathbf{o}| - 1 \end{cases} \tag{9}$$

Here $\circ$ represents the Hadamard (point-wise) matrix multiplication, $\boldsymbol{P}^\top$ denotes the transpose of the matrix $\boldsymbol{P}$, and $\mathbb{1}$ is a column vector of ones, and $\boldsymbol{\omega}_t$ is the coloumn vector that represents the label we are observing at time t of matrix $\boldsymbol{\omega}$. The resulting vectors $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ for each time step t are then related to $\alpha_s(t)$ and $\beta_s(t)$ for some s by:

$$\boldsymbol{\alpha}_t = \begin{bmatrix} \alpha_{s_0}(t) \\ \vdots \\ \alpha_{s_{|S|-1}}(t) \end{bmatrix}, \ \beta_t = \begin{bmatrix} \beta_{s_0}(t) \\ \vdots \\ \beta_{s_{|S|-1}}(t) \end{bmatrix} \tag{10}$$

We introduce $\boldsymbol{\alpha}_{ti}$ and $\boldsymbol{\beta}_{ti}$ as the combined matrix for α and β respectively, $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$.

$$\boldsymbol{\alpha}_{ti} = \begin{bmatrix} \alpha_{s_0 t_0} & \cdots & \alpha_{s_{|S|-1} t_{|\mathbf{o}|-1}} \\ \vdots & \ddots & \vdots \\ \alpha_{s_0 t_0} & \cdots & \alpha_{s_{|S|-1} t_{|\mathbf{o}|-1}} \end{bmatrix} \text{ and } \boldsymbol{\beta}_{ti} = \begin{bmatrix} \beta_{s_0 t_0} & \cdots & \beta_{s_{|S|-1} t_{|\mathbf{o}|-1}} \\ \vdots & \ddots & \vdots \\ \beta_{s_0 t_0} & \cdots & \beta_{s_{|S|-1} t_{|\mathbf{o}|-1}} \end{bmatrix} \tag{11}$$

γ and ξ can be expressed in terms of matrix operations as follows:

$$\boldsymbol{\gamma}_t = (\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\, \boldsymbol{\beta}_{ti}))^{-1} \cdot \boldsymbol{\alpha}_t \circ \boldsymbol{\beta}_t \tag{12}$$

$$\boldsymbol{\xi}_t = ((\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\, \boldsymbol{\beta}_{ti}))^{-1} \cdot \boldsymbol{P}) \circ (\boldsymbol{\alpha}_t \otimes (\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1})) \tag{13}$$

Here $\otimes$ represents the Kronecker (block) matrix multiplication, $\cdot$ denotes the scalar product and $^{-1}$ denotes the elementwise inverse of a matrix.

We can simplify $\sum_{i=1}^{|\mathbf{o}|-1}(\boldsymbol{\alpha}_{ti}\boldsymbol{\beta}_{ti})$ as, the sum does not depend on t:

$$\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\, \boldsymbol{\beta}_{ti}) = \sum_{i=1}^{|\mathbf{o}|-1} \boldsymbol{\alpha}_{|\mathbf{o}|-1 i} \tag{14}$$

$$= \mathbb{1}^T \boldsymbol{\alpha}_{|\mathbf{o}|-1} \tag{15}$$

Here $\mathbb{1}^T$ is a row vector of ones, and $\boldsymbol{\alpha}_{|\mathbf{o}|-1}$ is the last column of the matrix $\boldsymbol{\alpha}_{ti}$.

So we get:

$$\boldsymbol{\gamma}_t = (\mathbb{1}^T \boldsymbol{\alpha}_{|\mathbf{o}|-1})^{-1} \cdot \boldsymbol{\alpha}_t \circ \boldsymbol{\beta}_t \tag{16}$$

$$\boldsymbol{\xi}_t = ((\mathbb{1}^T \boldsymbol{\alpha}_{|\mathbf{o}|-1})^{-1} \cdot \boldsymbol{P}) \circ (\boldsymbol{\alpha}_t \otimes (\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1})) \tag{17}$$

The resulting vectors $\boldsymbol{\gamma}_t$ and $\boldsymbol{\xi}_t$ for each time step t are then related to $\gamma_s(t)$ and $\xi_{ss'}(t)$ for some s, s' by:

$$\boldsymbol{\gamma}_t = \begin{bmatrix} \gamma_{s_0}(t) \\ \vdots \\ \gamma_{s_{|S|-1}}(t) \end{bmatrix}, \boldsymbol{\xi}_t = \begin{bmatrix} \xi_{s_0 s_0}(t) & \cdots & \xi_{s_0 s_{|S|-1}}(t) \\ \vdots & \ddots & \vdots \\ \xi_{s_{|S|-1} s_0}(t) & \cdots & \xi_{s_{|S|-1} s_{|S|-1}}(t) \end{bmatrix} \tag{18}$$

We can update the parameters with matrix operations as follows:

$$\boldsymbol{P} = (\mathbb{1} \oslash \boldsymbol{\gamma}) \bullet \boldsymbol{\xi} \tag{19}$$

$$\boldsymbol{\omega}_s(o) = (\mathbb{1} \oslash \boldsymbol{\gamma}) \bullet (\sum_{t=1}^{|\mathbf{o}|-1} \boldsymbol{\gamma}_t \otimes \mathbb{1}_{yt}^{|\mathbf{o}|-1}) \tag{20}$$

$$\boldsymbol{\pi} = \boldsymbol{\gamma}_1 \tag{21}$$

Where $\oslash$ denotes Hadamard division (elementwise division) product and $\bullet$ denotes the Katri-Rao product (column-wise Kronecker product). In the formulas above, $\mathbb{1}$ denotes a column vector of ones, $\mathbb{1}_{yt}$ denotes a column vector that is $|\mathcal{L}|$ long, with all elements set to zero except for the element at index where $o_t = l$ which is set to one.

γ and ξ are the sum of the respective vectors over all time steps t:

$$\boldsymbol{\gamma} = \sum_{t=1}^{|\mathbf{o}|-1} \boldsymbol{\gamma}_t \text{ and } \boldsymbol{\xi} = \sum_{t=1}^{|\mathbf{o}|-1} \boldsymbol{\xi}_t \tag{22}$$

3 Implementation

In this section, we will discuss the implementation of the project. We will start by discussing ADDs and their advantages and disadvantages. How we transition from vectors and matrices to ADDs will be discussed. We will then discuss the CUDD library and the Storm model checker. Finally, we will discuss the implementation of the matrix operations using ADDs.

3.1 Algebraic Decision Diagrams

Algebraic Decision Diagrams (ADDs) generalize Binary Decision Diagrams (BDDs) [20]. Unlike BDDs which are limited binary values ({0,1}) [21], ADDs extend this framework by allowing terminal nodes to store arbitrary values, such as integers or real numbers. This flexibility makes them well-suited for applications like probabilistic model checking, optimization, and symbolic matrix operations.

In an ADD, each path from the root to a terminal node represents a unique variable assignment, with terminal nodes storing the associated function values. Their hierarchical structure supports compact, symbolic representations and efficient manipulation of large functions.

Advantages of ADDs:

- Compact Representation: ADDs exploit redundancies in data, providing a compact representation for many functions. Shared subgraphs reduce memory usage, especially when storing large, structured data, such as matrices.
- Canonical Form: For a fixed variable ordering, ADDs are canonical. This property simplifies equivalence checking and ensures consistency in symbolic manipulations.
- Efficient Operations: Operations such as addition and multiplication can be performed directly on ADDs, often avoiding explicit enumeration of the underlying data.

Disadvantages of ADDs:

- Variable Ordering Sensitivity: The size and efficiency of ADDs depend heavily on the chosen variable ordering. A suboptimal ordering can lead to exponentially larger diagrams, reducing performance.
- Overhead in Construction: Constructing an ADD involves significant overhead compared to simpler data structures like arrays or hash tables, particularly for small datasets, where the benefits of compactness may not be realized.
- Complexity for Certain Functions: While ADDs work well for structured data, highly unstructured or random data can result in large, inefficient diagrams. In such cases, other data structures may be more suitable.

ADDs can symbolically encode row and column indices as binary variables, enabling efficient storage and manipulation of matrix data. This approach is particularly advantageous for sparse or structured matrices, where patterns and redundancies can be exploited.

In this work, ADDs provide a foundation for symbolic matrix operations like addition, multiplication, and Kronecker product. Their compact representation and efficiency make them a powerful tool for handling complex, parameterized computations.

TABLE 1
Binary encoding of a vector V of size 4

Vector Index	Value	Binary Encoding
0	1	00
1	2	01
2	3	10
3	4	11

3.2 Transition to ADDs

The first step in the implementation is to transition from vectors and matrices to ADDs. This conversion leverages the compact and efficient representation of ADDs to perform operations symbolically.

To convert a vector into an ADD, we encode the indices as binary variables and the values as terminal nodes. It is important to note that all the variables in the ADD has to be consistent, that is the number of variables in the ADD has to be the same for all the nodes in the ADD.

When a matrix is represented as an ADD, the matrix has to be square, as the ADD representation requires a square matrix, if the matrix is not square, it has to be padded with zeros to make it square. This is because in our implementation, we use the CUDD library, which requires the matrices to be square to be represented as ADDs.

3.2.1 Vector to ADD

We will illustrate the conversion from a vector to an ADD using an example. Consider the following vector:

$$V = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

In an ADD, each layer corresponds to one binary variable (or bit) in the encoding of an index. For a vector of size n, where $n = 2^k$, the binary representation of the column indices requires k bits each. In the case of the vector V, the vector has 4 elements, so it requires $4 = 2^2$ bits to represent the indices.

We can create the binary representation of the indices by using the binary representation of the numbers from 0 to k.

The binary representation of the vector V entries is shown in Table 1.

The binary encodings determine the structure of the decision diagram, where each entry in the vector is stored as a terminal node. The paths to these nodes are dictated by the binary representation of their indices. The root node represents the first bit of the binary encoding, with subsequent layers corresponding to the remaining bits.

The ADD representation of the vector V is shown in Figure 1. For matrices represented as ADDs, the structure is similar to vectors, but with two sets of binary variables for row and column indices. The terminal nodes store the matrix entries, with paths determined by the binary representation of the row and column indices. The ADD variables are interleaved, with the row and column indices alternating in the path from the root to the terminal nodes. This is the way the CUDD library represents matrices as ADDs.

3.3 CUDD

The Colorado University Decision Diagram (CUDD) library [22] is a powerful tool for implementing and manipulating decision diagrams, including BDDs and ADDs. ADDs are compact representations of functions, often used to handle large state spaces symbolically and efficiently.

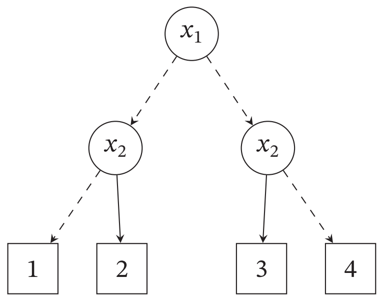

Fig. 1. ADD representation of a vector V of size 4

In this project, the CUDD library stores ADDs and performs operations on them. Its optimized algorithms and efficient memory management allow us to handle large and complex matrices symbolically, leading to significant performance improvements over traditional methods.

The CUDD library is implemented in C, ensuring high-performance execution, but it also ensures it can be used in C++ programs.

3.4 Storm

Storm is a versatile, open-source probabilistic model checking tool designed to verify the correctness and properties of stochastic models [8]. It supports a wide range of probabilistic models, including Markov Chains (MCs) and Markov Decision Processs (MDPs).

It does not include HMMs in its supported models, but a HMM can be enoded as an MC [23]. Storm allows users to analyze models efficiently by computing various quantitative properties, such as probabilities, expected rewards, or long-run averages.

The main purpose of this project is to provide data-driven modeling tools for formal verification. The Storm model checker is widely recognized as a state-of-the-art verification tool for probabilistic models, known for its efficiency and scalability. For this reason, in this project we have chosen integrate our symbolic implementation of the Baum Welch algorithm within Stom.

Storm uses the CUDD library for the symbolic representation of models, which makes it a natural choice for our implementation. This translate into working directly with the symbolic representation of the models provided by the parsers available in Storm that, in principle could be direcly used (off the shelf) by Storm for verification purposes.

3.5 Matrix operations using ADDs

The matrix operations are implemented using ADDs. The matrix operations implemented are matrix transpose, matrix addition, matrix multiplication, Hadamard product, Hadamard division, Kronecker product and Khatri-Rao product.

The matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

are used as examples in the following subsections.

In CUDD the *DdManager* is used to manage the ADDs and the operations on them. The *DdNode* is used to represent both the ADDs and the variables in the ADDs, that is the row and column indices in the ADDs.

3.5.1 Matrix Transpose

The matrix transpose is implemented by swapping the row and column variables in the ADD. Specifically, for each path in the ADD representing an entry (i, j), the roles of the row index i and column index j are exchanged. The terminal nodes (values of the matrix entries) remain unchanged. The transpose of matrix A is:

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

In CUDD the function we use for transposing an ADD is implemented as *Cudd_addSwapVariables(DdManager * dd, DdNode * f, DdNode ** x, DdNode ** y, int n)*, where f is the ADD to be transposed, x and y are the set variables to be swapped and n is the size of the variables to be swapped.

3.5.2 Matrix addition

Matrix addition is implemented by adding the terminal nodes of two ADDs while keeping the structure of the row and column indices consistent. The process involves:

1) Traversing the paths of both ADDs simultaneously.
2) Summing the values at the terminal nodes where the row and column indices match.

The resulting ADD represents the element-wise sum of the two matrices. The sum of matrices A and B is:

$$A + B = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

In CUDD the function we use for adding two ADDs is implemented as *Cudd_addApply(DdManager * dd, Cudd_addPlus(), DdNode * f, DdNode * g)*, where f and g are the two ADDs to be added and *Cudd_addPlus()* is the function that is used to add the two ADDs.

3.5.3 Matrix multiplication

Matrix multiplication is implemented symbolically using the dot product of the row and column indices. In the ADD:

1) For each pair of rows in the first matrix and columns in the second matrix, the corresponding elements are multiplied.
2) The products are summed along the shared index, combining them into the final terminal nodes of the resulting ADD.

The hierarchical structure of the ADD ensures that only relevant paths are explored, making the operation efficient. The product of matrices A and B is

$$A \times B = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

We use the function *Cudd_addMatrixMultiply(DdManager dd, DdNode A, DdNode B, DdNode **z, int nz)* in CUDD to multiply two ADDs. The function takes two ADDs A and B to be multiplied as input and returns a pointer to the resulting ADD. z is the set of variables that are dependent on the columns in A and the rows in B. A is assumed to have the same number of columns as B has rows, namely *nz*. In the implementation of the matrix multiplication, we needed to rename the variables in the ADDs to do the matrix multiplication, namely As columns variables and Bs row variables, as CUDD requires the variables in the ADDs to have the same name.

3.5.4 Hadamard product

The Hadamard product is implemented by pairwise multiplication of corresponding terminal nodes in the two ADDs. For each matching row-column index pair (i, j):

1) The values from both ADDs are multiplied.
2) The resulting product is stored in the terminal node of the new ADD.

The structure of the indices remains unchanged. The Hadamard product of matrices A and B is:

$$A \circ B = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

In CUDD the function we use for Hadamard product is implemented as *Cudd_addApply(DdManager * dd, Cudd_addTimes(), DdNode * f, DdNode * g)*, where f and g are the two ADDs to be multiplied and *Cudd_addTimes()* is the function that is used to multiply the two ADDs elementwise.

3.5.5 Hadamard division

The Hadamard division is implemented as Hadamard product, but with division instead of multiplication. See subsubsection 3.5.4 for more details. The Hadamard division of matrices A and B is:

$$A \oslash B = \begin{bmatrix} 0.2 & 0.3333 \\ 0.4286 & 0.5 \end{bmatrix}$$

The Hadamard division is implemented in CUDD by *Cudd_addApply(DdManager * dd, Cudd_addDivide(), DdNode * f, DdNode * g)*, where f and g are the two ADDs to be divided and *Cudd_addDivide()* is the function that is used to divide the two ADDs elementwise.

3.5.6 Kronecker product

The Kronecker product is implemented by symbolically expanding the indices and terminal nodes of the two matrices. This operation results in a new ADD with dimensions equal to the sum of the dimensions of the input matrices. The Kronecker product generalizes the outer product, multiplying each element of the first matrix by the entire second matrix.

For a specific entry in the first matrix A at row i and column j, with value a_{ij}, the second matrix B is scaled by a_{ij}, and its indices are adjusted as follows: and its indices are adjusted as follows:

1) Multiply the values of B by a_{ij}.
2) The row indices of B are shifted by $i \times \text{rows}(B)$ in the new ADD.
3) The column indices of B are shifted by $j \times \text{columns}(B)$ in the new ADD.
4) The scaled values are stored in the new ADD.

This operation is not natively supported in the CUDD library, so we implemented it explicitly as a custom function. The Kronecker product of matrices A and B is:

$$A \otimes B = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

3.5.7 Khatri-Rao product

The Khatri-Rao product is implemented by combining rows of the first matrix with corresponding rows of the second matrix. For each row index i:

1) The elements of row i in the first matrix are multiplied element-wise with the entire row i in the second matrix.
2) The resulting row is constructed symbolically within the ADD.

This operation is not natively supported in the CUDD library, so we implemented it explicitly as a custom function. The Khatri-Rao product of matrices A and B is:

$$A \bullet B = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

The resulting ADD has the dimensions of sum of the dimensions of the two matrices, as in the Kronecker product.

4 Class Diagram

The class diagram in Figure 2 shows the relationships between the different classes in the system. The `Model` class represents the underlying model of the system, which can be a CTMC, DTMC, HMM and MDP. The `Model` class has a aggregation relationship with the `Algorithm` class, which represents the functions used in the Baum-Welch algorithm, with and without ussing log-semiring. The `Algorithm` class has a dependency relationship with the `CUDD` class, which is a wrapper for the CUDD library. The `CUDD` class is used to perform the matrix operations as ADD's required by the Baum-Welch algorithm. The `Model` class also has a aggregation relationship with the `CUDD` class, as the `Model` class uses the `CUDD` class to perform the ADD operations.

4.1 `Model` Class

The `Model` class serves as the foundation for representing various probabilistic models like CTMC, MDP, and DTMC. It holds fields needed to describe these models, such as the transition matrices, emission probabilities, and initial states, all represented using Algebraic Decision Diagrams (ADDs). The `Model` class also provides methods for training models, such as the Baum-Welch algorithm.

Attributes:

- `Type_model`: Defines the type of model (e.g., CTMC, MDP, DTMC).
- `transfer`: A list of ADD structures representing state transition probabilities.
- `Emission`: An ADD for the emission probabilities (relevant in Hidden Markov Models).
- `pi`: The initial state distribution, also stored as an ADD.
- `training_set`: A collection of observed data.

Methods:

- `Instantiate_with_parameters(prismfile, parameters: Dictionary)`: Instantiates a model with specified parameters.
- `Instantiate_without_parameters(prismfile)`: Creates a model without additional parameters.
- `Baum-welsh(log, Model)`: This method implements the Baum-Welch algorithm for training Hidden Markov Models, utilizing various operations from the `Algorithm` class.

4.2 `CUDD` Class

The `CUDD` class (CUDD Manager) is responsible for managing ADDs. These ADDs are crucial in representing the probabilistic data structures used in the `Model` class. `CUDD` provides a set of operations that allow mathematical and logical manipulation of these diagrams.

Attributes:

- `rowvars, colvars`: Representing variables used in the ADD structures.
- `ADD`: The main data structure for storing probabilities or logical expressions.
- `Manager`: A control structure that coordinates operations on ADDs.

Methods:

- `Hadamard(), Log_Hadamard()`: Perform element-wise operations on ADDs.
- `Matrix_mul(), Log_matrix_mul()`: For matrix multiplications.
- `Sum(), Transpose()`: Additional helper methods for summing and transposing ADDs.

4.3 `Algorithm` Class

The `Algorithm` class encapsulates various methods for performing probabilistic calculations. These methods are mainly used for inference in models such as Hidden Markov Models (HMM) and Markov Chains.

Methods:

- `calculate_alpha(), calculate_beta()`: Compute the forward (alpha) and backward (beta) probabilities, respectively.
- `calculate_gamma(), calculate_xi()`: Intermediate probability calculations needed for parameter estimation and model training.

Each method operates on the ADD structures created and managed by the `CUDD` class, ensuring efficient computation of the probabilities.

4.4 Relationships Between Classes

4.4.1 `Model` to `Algorithm`: Association Relationship

The `Model` class uses the `Algorithm` class to compute the forward-backward probabilities and other values necessary for inference. The Baum-welsh(log, Model) method in `Model` invokes the relevant methods from `Algorithm` (calculate_alpha(), calculate_beta(), etc.) during the training process of HMMs. These methods, while called collectively in Baum-Welch, can also be used independently to perform specific calculations.

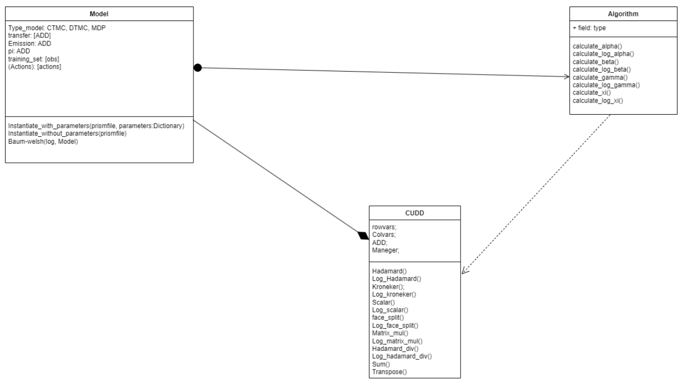

Fig. 2. Class diagram of the system

4.4.2 `Model` to `CUDD`: Aggregation Relationship

The `Model` class contains several attributes (transfer, Emission, pi) that are represented as ADDs, managed by the `CUDD` class. This relationship is best represented as an aggregation, where the `Model` holds instances of ADD but does not directly manage their internal workings. Instead, `CUDD` provides the operations required to manipulate and operate on these diagrams, such as matrix multiplication or element-wise functions (Hadamard products). The `Model` depends on `CUDD` for these operations, making it an integral part of the system's backend.

4.4.3 `Algorithm` to `CUDD`: Dependency Relationship

The `Algorithm` class depends on the `CUDD` class for all its operations on ADDs. Every method in `Algorithm` (e.g., calculate_alpha(), calculate_gamma()) relies on ADD operations provided by `CUDD`, such as Matrix_mul() and Hadamard(). This is represented by a dependency relationship, where `Algorithm` calls `CUDD`'s methods to perform its computations.

ACRONYMS

AAU	Aalborg University. 1
ADD	Algebraic Decision Diagram. 1, 2, 5–8
BDD	Binary Decision Diagram. 5, 6
HMM	Hidden Markov Model. 2–4, 6
MC	Markov Chain. 6
MDP	Markov Decision Process. 6

REFERENCES

[1] A. A. Markov and J. J. Schorr-Kon, *Theory of algorithms*. Springer, 1962.

[2] P. Milazzo, "Analysis of covid-19 data with prism: Parameter estimation and sir modelling," in *From Data to Models and Back*, Springer International Publishing, 2021, pp. 123–133, ISBN: 978-3-030-70650-0.

[3] F. Ciocchetta and J. Hillston, "Bio-pepa: A framework for the modelling and analysis of biological systems," *Theoretical Computer Science*, vol. 410, no. 33-34, pp. 3065–3084, 2009.

[4] R. S. Mamon and R. J. Elliott, *Hidden Markov models in finance*. Springer, 2007, vol. 4.

[5] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[6] E. M. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, Springer, 1997, pp. 54–56.

[7] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 585–591.

[8] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker storm," *International Journal on Software Tools for Technology Transfer*, pp. 1–22,

[9] G. Bacci, A. Ingólfsdóttir, K. G. Larsen, and R. Reynouard, "Mm algorithms to estimate parameters in continuous-time markov chains," 2023. arXiv: 2302.08588 `[cs.LG]`.

[10] P. Kenny, T. Stafylakis, P. Ouellet, V. Gupta, and M. J. Alam, "Deep neural networks for extracting baum-welch statistics for speaker recognition.," in *Odyssey*, vol. 2014, 2014, pp. 293–298.

[11] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi, "An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition," *Bell System Technical Journal*, vol. 62, no. 4, pp. 1035–1074, 1983.

[12] R. I. Davis and B. C. Lovell, "Comparing and evaluating hmm ensemble training algorithms using train and test and condition number criteria," *Formal Pattern Analysis & Applications*, vol. 6, pp. 327–335, 2004.

[13] R. Reynouard, A. Ingólfsdóttir, and G. Bacci, "Jajapy: A learning library for stochastic models," in *International Conference on Quantitative Evaluation of Systems*, Springer, 2023, pp. 30–46.

[14] L. E. Hansen, C. Ståhl, D. R. Petersen, S. Aaholm, and A. M. Jakobsen, "Symbolic parameter estimation of continuous-time markov chains," *AAU Student Projects*, 2023. eprint: https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/a7me0f/alma9921651457905762.

[15] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The annals of mathematical statistics*, vol. 37, no. 6, pp. 1554–1563, 1966.

[16] R. S. Chavan and G. S. Sable, "An overview of speech recognition using hmm," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 6, pp. 233–238, 2013.

[17] V. De Fonzo, F. Aluffi-Pentini, and V. Parisi, "Hidden markov models in bioinformatics," *Current Bioinformatics*, vol. 2, no. 1, pp. 49–61, 2007.

[18] H. Murveit and R. Moore, "Integrating natural language constraints into hmm-based speech recognition," in *International Conference on Acoustics, Speech, and Signal Processing*, IEEE, 1990, pp. 573–576.

[19] B. Benyacoub, I. ElMoudden, S. ElBernoussi, A. Zoglat, and M. Ouzineb, "Initial model selection for the baum-welch algorithm applied to credit scoring," in *Modelling,*

Computation and Optimization in Information Systems and Management Sciences: Proceedings of the 3rd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences-MCO 2015-Part II, Springer, 2015, pp. 359–368.

[20] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebric decision diagrams and their applications,” *Formal methods in system design*, vol. 10, pp. 171–206, 1997.

[21] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[22] F. Somenzi, “Cudd: Cu decision diagram package,” *Public Software, University of Colorado*, 1997.

[23] L. R. Rabiner, “A tutorial on hidden markov models and selected applications in speech recognition,” *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

Appendix A
Compiling in draft

You can also compile the document in draft mode. This shows todos, and increases the space between lines to make space for your supervisors feedback.