



On learning stochastic models: from theory to practice

by
Raphaël Reynouard

Dissertation submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
November 14, 2023

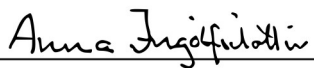

Thesis Committee:

Supervisors: Anna Ingólfssdóttir, Professor,
Reykjavík University, Reykjavík, Iceland
Giovanni Bacci, Associate Professor,
Aalborg University, Aalborg, Denmark

Examiners : Nils Jansen, Professor,
Ruhr University, Bochum, Germany
Catia Trubiani, Associate Professor,
Gran Sasso Science Institute, L'Aquila, Italy

© Raphaël Reynouard
November 2023

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

	Date of Signature
 _____ Anna Ingólfssdóttir Professor, Reykjavík University, Reykjavík, Iceland	<u>21.11.2023</u>
 _____ Giovanni Bacci Associate Professor, Aalborg University, Aalborg, Denmark	<u>21.11.2023</u>
 _____ Nils Jansen Professor, Ruhr University, Bochum, Germany	<u>27.11.2023</u>
 _____ Catia Trubiani, Associate Professor, Gran Sasso Science Institute, L'Aquila, Italy	<u>27.11.2023</u>

REYKJAVIK UNIVERSITY

On learning stochastic models: from theory to practice

Raphaël Reynouard

November 14, 2023

Abstract

The field of model checking offers numerous tools for analysing stochastic models. This analysis provides a comprehensive understanding of the behaviours exhibited by the system represented in the model. Consequently, such analyses are of paramount importance for critical systems. Nonetheless, in certain application domains, the model is not readily accessible and needs to be acquired from partially-observable executions of the system under analysis.

This thesis proposes to improve the learning of stochastic models, thereby facilitating the application of model checking to systems for which no model is currently available. This objective is realised via three discrete strategies: (i) formulating an active learning algorithm to learn Markov decision processes, (ii) devising a learning algorithm tailored for synchronised compositions of continuous-time Markov chains, and (iii) developing a library compatible with model checkers, streamlining the process of stochastic model acquisition and its seamless incorporation into the model checking procedure. The first two strategies focus on enhancing and extending the theoretical foundations of learning stochastic models, while the third one centers on facilitating the application of learning stochastic models and their integration into the model checking workflow.

Setningarfræðilegar aðferðir við neikvæðar niðurstöður í algebrum vinnslu og mótalogík

Raphaël Reynouard

November 14, 2023

Útdráttur

Fagsviðið sem fjallar um könnun líkana býður upp á margs konar tól til að kanna líkindafræðileg líkön. Slíkar kannanir skapa gagnlegt innsæi í hegðun þess kerfis sem líkanið lýsir. Slíkar greiningar eru sérlega mikilvægar fyrir krítísk kerfi. Þrátt fyrir þetta er í mörgum tilfellum ekki auðvelt að fá aðgang að slíkum líkönum og því þörf á að geta skapað þau frá grunni með því að keyra kerfin og safna þeim upplýsingum sem eru aðgengilegar.

Þessi ritgerð býður upp á leiðir til að læra líkindafræðileg líkön og á þann hátt gera það mögulegt að nota þessa aðferðafræði í tilfellum þar sem engin líkön hafa verið til fram að þessu. Þetta felur í sér eftirfarandi þrjár nálganir: (i) Að setja fram virkt reiknirit til að læra Markov ákvörðunarferli (ii) Að setja fram lærdómsreiknirit fyrir samstillta samsetningu á Markov-keðjum með samfelldan tíma, og (iii) að þróa hugbúnaðarkerfi sem er samhæft helstu kerfum sem sjá um könnun líkana og gera þannig allt ferlið meira straumlínulagað.

Fyrstu tvær aðferðirnar miða fyrst og fremst að því að efla fræðilegan bakgrunn fyrir lærdómsreiknirit sem læra líkindafræðileg líkön en markmið hins þriðja gengur út á að bjóða upp á hugbúnað sem bætir almennt flæðið í lærdóms- og könnunarferlinu.

Acknowledgements

This thesis would not have been possible without the assistance of several individuals, whom I would like to express my heartfelt gratitude to.

I would like to commence by thanking my supervisors, Anna and Giovanni, for their guidance, mentorship, both in the field of science and academia, and for their unwavering patience.

Similarly, I would like to extend my appreciation to the Department of Computer Science at the Háskólinn í Reykjavík (HR, or RU in english), not only for their exemplary administrative support and the enlightening lunch talks organised by their dean, Luca Aceto but also for the encouragement and support provided by fellow PhD students in the department, particularly Shalini, Benedikt and Stian.

This thesis also owes a debt of gratitude to Kim Larsen for his invaluable advice and warm welcome. I will forever hold dear the memory of my three-week stay at Aalborg University (AAU), as well as the winter evening in 2022 when we gathered to watch France triumph over Denmark in the World Cup.

I would also like to express my thanks to the professors that placed their trust in me and allowed me to present my work at their University: Joost-Pieter Katoen at Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), and Jean-François Raskin at the Université Libre de Bruxelles (ULB).

Finally, this thesis would not have come to fruition without the invaluable guidance provided by the numerous individuals I had the privilege to meet during these three years. Therefore, I would like to extend my gratitude to Tim and Mohammadreza for their time and insights.

This work has been supported by the *Learning and Applying Probabilistic Systems* project (Project nr. 206574-051) funded by Rannis.

List of Tables

4.1	Formalisms and algorithms supported by AALPY	42
5.1	L_{MDP}^* vs Active-BW on the 3x3 grid world model.	55
5.2	L_{MDP}^* vs Active-BW on the 4x4 grid world model.	55
6.1	Performance comparison on selected QComp benchmarks [1].	67
6.2	Parameter estimation on the approximated SIR model.	70
7.1	Key characteristics of the selected learning algorithms for Markov models.	78
7.2	Results for an unbiased Yao-Knuth's die ($p = 0.5$).	82
7.3	Results for a biased Yao-Knuth's die ($p = 0.9$).	82
7.4	Results for the comparison of JAJAPY and HMMLEARN.	84
7.5	Learning algorithms for Markov models implemented in JAJAPY and AALPY.	85
7.6	Results for an unbiased Yao-Knuth's die ($p = 0.5$).	86
7.7	Results for a biased Yao-Knuth's die ($p = 0.9$).	86
7.8	Results for the 3x3 deterministic grid-world model.	87
7.9	Results for the 4x4 non-deterministic grid-world model.	87

List of Figures

2.1	A simple MC with three states	8
2.2	MDP model for a Shifumi game	9
2.3	A simple CTMC with three states	11
2.4	A non-deterministic MC.	13
2.5	Three slightly different MCs.	16
2.6	An HMM for inferring outdoor temperature based on headwears. . .	17
2.7	MC induced by the HMM from Example 2.5.1.	18
3.1	State merging approach workflow [2]	33
3.2	Example PTA	33
3.3	An example of state merging	34
3.4	An example of TPTA	35
3.5	An example of IOPTA	36
3.6	Interactions between the SUL, the teacher and the learner.	38
4.1	Simple execution of AALPY to learn an MC using Alergia.	41
5.1	Active Sampling Strategy	52
5.2	Grid world models.	53
5.3	An empirical comparison of Active-BW and BW	54
6.1	SIR model	58
6.2	Prism model for the tandem queueing network	63
6.3	Synchronous composition of two pCTMCs	63
6.4	Empirical evaluation of our parameter estimation technique	68
6.5	Approximated SIR model.	69
7.1	A complete modeling and verification workflow using JAJAPY	76
7.2	Simple execution of JAJAPY BW to learn an MC with 10 states. . .	76
7.3	Class diagram for JAJAPY model classes.	79
7.4	The Yao-Knuth's die from [3]	81
7.5	Empirical evaluation of JAJAPY performance	83
7.6	An HMM with 5 states.	84
7.7	Two grid worlds models	86
C.1	Empirical evaluation of JAJAPY performance on a bigger training set	110

Contents

Abstract	iv
List of Tables	ix
List of Figures	xi
1 Introduction	1
2 Preliminaries	5
2.1 A short history of model checking.	5
2.2 Notations	6
2.3 Markov Models	7
2.3.1 Markov Chains	7
2.3.2 Markov Decision Processes	8
2.3.3 Continuous-Time Markov Chains	10
2.4 Probability Measure of a Markov Model	12
2.4.1 Probability Measure of discrete-time models	12
2.4.2 Probability Measure of continuous-time models	14
2.4.3 Measuring the difference between models	15
2.5 Few words about Hidden Markov Models	15
2.5.1 The classic Hidden Markov Models	15
2.5.2 Gaussian Hidden Markov Models	18
I State of the Art	21
3 Methods	25
3.1 The Expectation-Maximisation approach	27
3.1.1 Baum-Welch for MCs	29
3.1.2 Baum-Welch for MDPs	30
3.1.3 Baum-Welch for CTMCs	31
3.2 The State-Merging approach	33
3.2.1 Alergia	34
3.2.2 TAlergia	35
3.2.3 IOAlergia	36
3.2.4 MDI	37
3.3 Learning with Queries	37

4	Tools	41
4.1	AALPY	41
4.2	Others	42
4.3	Conclusion	43
II	Contributions	45
5	Active Baum-Welch for MDPs	49
5.1	Introduction	49
5.2	Preliminaries	50
5.3	The Active-BW algorithm	51
5.4	Experimental results	52
5.4.1	Active-BW vs BW	52
5.4.2	Active BW vs L_{MDP}^*	54
5.5	Conclusions and Future Work	56
6	Parameter estimation of pCTMCs	57
6.1	Introduction	57
6.2	Preliminaries and Notation	61
6.3	Parametric Continuous-time Markov chains	61
6.4	Estimating Parameters from Partial Observations	64
6.5	Experimental evaluation	67
6.6	Case Study: SIR modelling of pandemic	69
6.7	Conclusion and Future Work	70
7	Jajapy	73
7.1	Introduction	73
7.2	Jajapy in a nutshell	75
7.3	Learning probabilistic models	77
7.4	Architecture and technical aspects	78
7.5	Experimental evaluation and comparison	80
7.5.1	JAJAPY validation testing	81
7.5.2	Experimental evaluation of the scalability	82
7.5.3	Comparison with HMMLEARN	84
7.5.4	Comparison with AALPY	85
7.6	Conclusions and Future Work	88
III	Epilogue	89
8	Closing Remarks	91
A	The Baum-Welch algorithm in details	93
A.1	Convergence of the EM algorithm	93
A.2	Baum-Welch for MCs	95
A.3	Baum-Welch for MDPs	98

A.4 Baum-Welch for CTMCs	100
B Proof of Theorem 6.4.1	105
C Scalability evaluation over training set size	109
Bibliography	111

Chapter 1

Introduction

The world around us is becoming increasingly populated by interconnected devices. Think about our cell phones, computers, and even items like cars, supermarket checkouts, ATMs, traffic lights, and hotel room locks. You can find digital convenience everywhere – from the safety gate at the pool to the subway. In this digital age, everything is seamlessly connected, making life smoother and more enjoyable. No more mailing in forms or waiting in long lines for train tickets. The transformation of our lives through digitisation and automation is well underway. We're now talking about concepts like smart cities and connected homes, where everything from heating and lighting to locks and blinds can be managed through a simple smartphone app. From 2010 to 2019, the number of connections has increased from 8.8 billion to 20 billion, a compound annual growth rate of 10% [4]. These connected devices range from the most trivial aspects of our lives, such as lighting or entertainment, to the most critical, such as aviation or even health.

In the ever-evolving landscape of software development, ensuring the reliability and correctness of complex systems has become a paramount concern. Bugs and errors in software can lead to disastrous consequences, ranging from financial losses to compromised security and even loss of life. For instance, due to a race condition error, the Therac-25 Radiation Therapy Machine sometimes gave its patients radiation doses that were hundreds of times greater than normal between 1985 to 1987 [5]. As another example, software issues related to the Maneuvering Characteristics Augmentation System (MCAS) played a role in two deadly crashes in which 346 people died, involving Boeing 737 MAX planes: Lion Air Flight 610 on October 29, 2018, and Ethiopian Airlines Flight 302 on March 10, 2019. The system misinterpreted sensor data, causing the aircraft to nosedive uncontrollably. Boeing 737 MAX were subsequently grounded worldwide from March 2019 to November 2020. The accidents and subsequent grounding incurred Boeing approximately \$20 billion in fines, compensation, and legal expenses by 2020. Moreover, the ripple effects led to over \$60 billion in indirect losses due to the cancellation of around 1,200 orders [6].

To address these challenges, the field of formal verification, particularly model

checking, has emerged as a powerful technique for exhaustively verifying the correctness of software systems.

Definition 1.0.1 (Model Checking). *Model checking is an automated technique that, given a finite state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [7].*

In this context, a model is an abstraction of the actual system, capturing its essential behavioural aspects. The process involves exploring all possible states of the model and verifying if they adhere to specified properties, such as safety, liveness, deadlock, and more.

Model checking can be used for bug detection, offering a systematic approach to exhaustively analyse all possible states, helping to uncover bugs that might remain hidden in testing. In particular, model checking can be applied in the early design phases to catch issues before propagating deeper into the development process. Model checking can identify potential safety violations and security vulnerabilities before these systems are deployed and therefore ensure correct behaviour which is of utmost importance for safety-critical systems, such as those used in medical devices, automotive control, and aerospace. Being widely used for verifying the correctness of communication protocols, model checking ensures that systems exchange information accurately and securely. Finally, when a property is violated, model checking tools can provide counterexamples demonstrating how the violation occurs. This information can help developers to understand and rectify the issues.

Model checking uses various kinds of formal models to abstract real systems. Among them are Markov models, a class of mathematical frameworks, that provide a simple yet powerful way to analyse systems that evolve over time with probabilistic transitions. These models are named after Russian mathematician Andrey Markov. His work in the late 19th and early 20th centuries laid the foundation for what would become Markov models. His research focused on probability theory and stochastic processes, leading to the development of the concept of Markov chains. Markov chains describe the behaviour of systems where future states depend only on the current state, rendering them memoryless and suitable for modelling diverse scenarios. Markov models are used in finance, biology, telecommunications, natural language processing, robotics and control system, to name a few.

The research problem. Model checking tools typically operate under the presumption that the model is known. However, in various application domains, this assumption proves overly stringent. Frequently, the model is either inaccessible or, at most, only partially discernible. In such scenarios, the model is generally inferred empirically from a collection of partially observable executions, often called traces.

Employing machine learning to acquire Markov models offers a promising approach to overcome this challenge. By leveraging large sets of traces, machine learning algorithms can discern underlying patterns and transitions, ultimately constructing Markov models that approximate the system's behaviour. However, this task is not without its complexities. The curse of dimensionality, where the

number of possible states grows exponentially with system complexity, can hinder accurate modelling. Additionally, handling noise, incomplete data, and ensuring the model’s generalisability are persistent difficulties. Lastly, the process of comparing two Markov models, which is useful to assess their performance and select the most effective one for a specific task, proves to be a challenging endeavour. Consequently, assessing a learning algorithm’s performance or conducting a comparison between two distinct learning algorithms for Markov models can present difficulties.

Objectives. The goal of this thesis is to address the gap between machine learning and model checking by advancing the current state-of-the-art in stochastic model learning. This goal will be accomplished through three distinct approaches:

1. by creating an active learning algorithm to learn Markov decision processes,
2. by designing an algorithm for the estimation of parameters values in a continuous-time Markov chain modelled in PRISM based on a collection of partially-observable executions¹.
3. by developing a novel library that is seamlessly compatible with state-of-the-art model checkers (PRISM and STORM) facilitating the integration of stochastic model learning into the classical model checking workflow.

Outline. The document commences with an introductory chapter that elucidates the fundamental concepts and terminologies employed throughout this thesis. Following this introduction, the document is divided into three distinct parts: (i) State of the Art, it offers a survey of contemporary techniques for learning Markov models. It helps readers to understand the current landscape in this field, thus enabling them to appreciate the novel contributions presented in the thesis. (ii) Contributions, each of the three primary objectives defined at the outset of the document is elaborated upon in individual Chapters. These Chapters go deeper into the specific advancements and innovations introduced by the thesis, providing detailed insights into each objective. (iii) Epilogue, it synthesises and summarises the key findings, contributions, and future development of the thesis. This structured outline guides the reader through a logical progression of concepts, from foundational knowledge to the novel contributions and, ultimately, to a comprehensive conclusion.

1. We decided to use the PRISM modelling language since it is the most popular language to express compositions of models.

Chapter 2

Preliminaries

This chapter introduces most of the notations used in this thesis. For terminology not explicitly shown and for a deeper explanation, we refer to [7].

2.1 A short history of model checking.

The roots of model checking trace back to the 1970s with the introduction of automata theory and temporal logic into the formal verification landscape. Researchers like Edmund Clarke and E. Allen Emerson formulated the basic ideas of model checking, envisioning a method that would systematically explore all possible states of a system to verify desired properties [8, 9]. They also introduced temporal logic, a powerful language for expressing properties over time.

The '80s and '90s saw the creation of two pivotal model checking tools: $Mur\phi$, developed by David Dill's team at Stanford University [10], and SPIN, conceived by Gerard Holzmann at Bell Labs [11]. These tools demonstrated the practicality of model checking and its ability to uncover subtle errors in complex designs.

As systems grew larger and more complex, the challenge of state explosion prompted the development of symbolic model checking techniques. This era saw the emergence of BDDs (Binary Decision Diagrams) and SAT solvers (Satisfiability solvers), enabling efficient representation and manipulation of state spaces. This innovation greatly expanded the scope of model checking to handle larger designs. However, these tools support non-stochastic models only, as Kripke structures or finite automata, which are not sufficient to describe the complexity of current systems. In 2000, Marta Kwiatkowska's team released PRISM, a probabilistic and symbolic model checker [12], supporting stochastic models. Sixteen years later, the model checker STORM was released by Joost-Pieter Katoen's team at RWTH [13]. On the one hand, the utilisation of PRISM remains predominant and its language has become a modelling standard. On the other hand, STORM stands out for its superior efficiency in both spatial and temporal dimensions [14]. As we progress into the 2020s, model checking continues to evolve alongside technological advancements. Machine learning and AI are being explored to enhance the scalability and automation of verification processes. Furthermore, there's a growing emphasis on

model checking for software security, aiming to identify vulnerabilities and prevent potential cyberattacks.

2.2 Notations

We denote by Σ^n , Σ^* and, Σ^ω respectively the set of words of length $n \in \mathbb{N}$, finite length, and infinite length, built over the finite alphabet Σ .

A prefix of a (possibly finite) word $w = a_1a_2\dots$ is a finite word $w' = a_1a_2\dots a_i$ with $i \in \mathbb{N}_{>0}$ (if w is a finite word of length n we have $i \leq n$). We denote by $\text{pref}(w)$ the set of prefixes of a word w . We use also $\text{pref}(W)$ to denote the set of prefixes of the set of words W , i.e. $\text{pref}(W) = \bigcup_{w \in W} \text{pref}(w)$.

We denote by $w_1 \cdot w_2$ the concatenation of the two words w_1 and w_2 .

We denote by \mathbb{R} and \mathbb{N} respectively the sets of real and natural numbers, and by \mathbb{B} the Boolean domain. We denote by $|\Omega|$ the cardinality of a set Ω .

Probability theory. To begin, we briefly review some classical notions of measure and probability from [7] and [15]. These concepts are assumed to be familiar to the reader. We will use these notions to describe the probability of the possible executions of models.

Definition 2.2.1 (σ -algebra). *A σ -algebra is a pair (Ω, \mathcal{F}) where Ω is a nonempty set and $\mathcal{F} \subseteq 2^\Omega$ is a set of subsets of Ω containing the empty set and being closed under complement and countable unions, i.e.:*

1. $\emptyset \in \mathcal{F}$,
2. if $F \in \mathcal{F}$, then $\overline{F} = \Omega \setminus F \in \mathcal{F}$, and
3. if $F_1, F_2, \dots \in \mathcal{F}$, then $\bigcup_{n \geq 1} F_n \in \mathcal{F}$.

In light of the first two conditions, we know that \mathcal{F} contains necessarily Ω . Additionally, due to the third condition, \mathcal{F} must be closed under countable intersections, since

$$\bigcap_{n \geq 1} F_n = \overline{\bigcup_{n \geq 1} \overline{F_n}}.$$

A *probability measure* on (Ω, \mathcal{F}) is a function $Pr : \mathcal{F} \rightarrow [0, 1]$ such that $Pr(\Omega) = 1$, and for $(F_n)_{n \geq 1}$ any family of pairwise disjoint events $F_n \in \mathcal{F}$:

$$Pr\left(\bigcup_{n \geq 1} F_n\right) = \sum_{n \geq 1} Pr(F_n). \quad (2.1)$$

A *probability space* is a σ -algebra associated with a probability measure: a probability space is therefore a triplet $(\Omega, \mathcal{F}, Pr)$.

Let Ω be a countable set. A function $\mu : \Omega \rightarrow [0, 1]$ such that $\sum_{\omega \in \Omega} \mu(\omega) = 1$ induces a probability measure $Pr_\mu : 2^\Omega \rightarrow [0, 1]$ on the discrete σ -algebra 2^Ω as follows: for any $F \subseteq \Omega$ subset of Ω , $Pr_\mu(F) = \sum_{\omega \in F} \mu(\omega)$. We say that μ is a

probability *distribution* on Ω . In the following, we abbreviate $Pr_\mu(F)$ by $\mu(F)$, and we denote by $\mathcal{D}(\Omega)$ the set of discrete probability distributions on Ω .

For a proposition p , we write $\llbracket p \rrbracket$ for the Iverson bracket of p , i.e., $\llbracket p \rrbracket = 1$ if p is true, otherwise 0.

2.3 Markov Models

In this section, we present different classic model formalisms to describe behaviours of systems. These models are essentially directed graphs, where nodes represent *states* of the system, and edges represent *transitions*, i.e. state-changes. Each state is associated to a *label* describing the meaning of this state. For instance, the states of a model describing a pedestrian traffic light could be labelled by ‘green’ or ‘red’.

The size of a Markov model \mathcal{M} is denoted by $|\mathcal{M}|$ and corresponds to the number of states in this model, i.e. the cardinality of its state space.

Remark 2.3.1. *In the literature, the size of a Markov model is sometimes defined as the sum of the size of each of its components, while here we consider the cardinality of its state space only.*

The transitions are defined differently for each formalism.

2.3.1 Markov Chains

In a Markov chain (MC), the next state is chosen according to a probability distribution over the states. This probability distribution depends on the current state only.

Definition 2.3.1. *A Markov chain is a tuple $\langle S, \mathcal{L}, \ell, \tau, \pi \rangle$ where*

1. S is a finite nonempty set of states,
2. \mathcal{L} is a finite nonempty set of labels,
3. $\ell : S \rightarrow \mathcal{L}$ is a labelling function which assigns a label to each state,
4. $\tau : S \rightarrow \mathcal{D}(S)$ is the transition function: the model moves from state s to s' with probability $\tau(s)(s')$, and
5. $\pi \in \mathcal{D}(S)$ is the initial distribution: the model starts in state s with probability $\pi(s)$.

Intuitively, \mathcal{M} starts in state s with probability $\pi(s)$. Then, it emits the label $\ell(s)$ and moves to state s' with probability $\tau(s)(s')$. In this sense, \mathcal{M} can be thought of as a state-machine emitting a sequence of labels.

Remark 2.3.2 (Absorbing state). *A state s is called absorbing if the model cannot leave it, i.e. $\tau(s)(s) = 1$.*

Remark 2.3.3 (Initial state). *π may be a Dirac distribution concentrated in one state s . In such case, we can use $s_{init} = s$ instead of π to describe the initial distribution.*

Example 2.3.1. Let $\mathcal{M} = \langle S, \mathcal{L}, \ell, \tau, \pi \rangle$ be a Markov chain where:

- $S = \{s_1, s_2, s_3\}$,
- $\mathcal{L} = \{\text{sunny, cloudy, rainy}\}$,
- $\ell(s_1) = \text{sunny}$, $\ell(s_2) = \text{cloudy}$ and $\ell(s_3) = \text{rainy}$,
- $\pi(s_1) = 1.0$ and $\pi(s_2) = \pi(s_3) = 0.0$, and
- $\tau(s_1)(s_1) = 0.8$, $\tau(s_1)(s_2) = 0.2$,
 $\tau(s_2)(s_1) = 0.2$, $\tau(s_2)(s_2) = 0.6$, $\tau(s_2)(s_3) = 0.2$,
 $\tau(s_3)(s_2) = 0.3$ and $\tau(s_3)(s_3) = 0.7$.

\mathcal{M} could be depicted as in Figure 2.1. □

Paths and traces

A path is a sequence in $\mathbf{Paths}^{\mathcal{M}} \subseteq S^\omega$ representing an infinite execution of an MC \mathcal{M} , and we denote by $\mathbf{Paths}_{\text{fin}}^{\mathcal{M}} \subseteq S^*$ the set of finite paths.

Analogously, we define a *trace* as a sequence of labels in $\mathbf{Traces}^{\mathcal{M}} \subseteq \mathcal{L}^\omega$ representing the emission of an infinite execution of \mathcal{M} , and we denote by $\mathbf{Traces}_{\text{fin}}^{\mathcal{M}} \subseteq \mathcal{L}^*$ the set of prefixes of $\mathbf{Traces}^{\mathcal{M}}$, i.e. the set of finite traces.

When the context is clear, we write \mathbf{Paths} , $\mathbf{Paths}_{\text{fin}}$, \mathbf{Traces} and $\mathbf{Traces}_{\text{fin}}$ respectively instead of $\mathbf{Paths}^{\mathcal{M}}$, $\mathbf{Paths}_{\text{fin}}^{\mathcal{M}}$, $\mathbf{Traces}^{\mathcal{M}}$ and $\mathbf{Traces}_{\text{fin}}^{\mathcal{M}}$.

Finally, we define $|\rho|$ the length of a finite path $\rho \in \mathbf{Paths}_{\text{fin}}^{\mathcal{M}}$ and $|o|$ the length of a finite trace $o \in \mathbf{Traces}_{\text{fin}}^{\mathcal{M}}$.

2.3.2 Markov Decision Processes

In Markov decision processes (MDPs), the next state is chosen according to a probability distribution depending on the current state and an input *action* given to the model. Hence, MDPs are *reactive* variants of MCs, since they reacts to streams of input actions.

Definition 2.3.2. A Markov decision process is a tuple $\langle S, \mathcal{L}, \ell, A, \{\tau_a\}_{a \in A}, \pi \rangle$ where S, \mathcal{L}, ℓ and π are defined as before, and

1. A is a finite nonempty set of actions,
2. $\tau_a: S \rightarrow \mathcal{D}(S)$ is a probabilistic transition function: the model moves from state s to s' while a is received as input action with probability $\tau_a(s)(s')$.

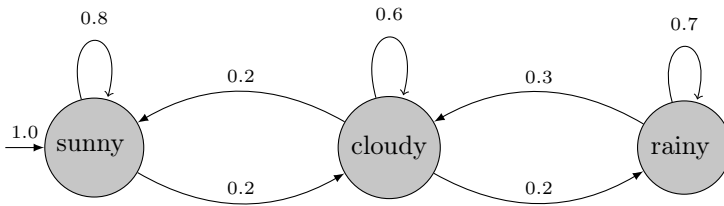


Figure 2.1 – A simple MC with three states

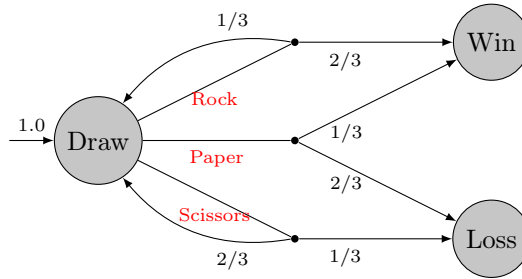


Figure 2.2 – An MDP modelling a Shifumi game where the opponent chooses Rock with probability $1/3$ and Scissors with probability $2/3$.

Intuitively, \mathcal{M} starts in state s with probability $\pi(s)$. Then, it emits the label $\ell(s)$ and, if it receives an input action $a \in A$, moves to state s' with probability $\tau_a(s)(s')$. In this sense, \mathcal{M} can be thought of as a state-machine that reacts to a sequence of n actions by emitting a sequence of $n + 1$ labels (the first label corresponds to the initial state, then n labels are emitted for the n input actions).

Remark 2.3.4 (Absorbing state). *A state s is called absorbing if the model cannot leave it, i.e. $\forall a \in A : \tau_a(s)(s) = 1$. Such transitions can be omitted in graphic representations, as in Figure 2.2.*

Remark 2.3.5 (Initial state). *π may be a Dirac distribution concentrated in one state s . In such case, we can use $s_{init} = s$ instead of π to describe the initial distribution.*

Example 2.3.2. Let $\langle S, \mathcal{L}, \ell, A, \{\tau_a\}_{a \in A}, \pi \rangle$ be an MDP modelling a game of Shifumi. This game stops once one of two player wins a round. Here, we consider that the opponent plays always ‘Rock’ with probability $1/3$ and ‘Scissors’ with probability $2/3$. We have

- $S = \{s_1, s_2, s_3\}$,
- $\mathcal{L} = \{\text{draw}, \text{win}, \text{loss}\}$,
- $\ell(s_1) = \text{draw}$, $\ell(s_2) = \text{win}$ and $\ell(s_3) = \text{lose}$,
- $\pi(s_1) = 1.0$ and $\pi(s_2) = \pi(s_3) = 0.0$,
- $A = \{\text{Rock}, \text{Paper}, \text{Scissors}\}$ and
- $\tau_{\text{Rock}}(s_1)(s_1) = 1/3$, $\tau_{\text{Rock}}(s_1)(s_2) = 2/3$,
 $\tau_{\text{Paper}}(s_1)(s_2) = 1/3$, $\tau_{\text{Paper}}(s_1)(s_3) = 2/3$,
 $\tau_{\text{Scissors}}(s_1)(s_1) = 2/3$, $\tau_{\text{Scissors}}(s_1)(s_3) = 1/3$.

\mathcal{M} could be depicted as in Figure 2.2. □

Remark 2.3.6 (MC and MDP). *An MC \mathcal{M} can be viewed as an MDP with only one action which is implicitly given to \mathcal{M} at each transition step.*

Paths and traces

As for MCs, we introduce the notion of *path* and *trace* for MDPs. The main difference consists in the fact that MDP traces and paths include actions.

An MDP path is a sequence in $\mathbf{Paths} \subseteq (S \times A)^\omega$ representing an infinite execution of an MDP, and we define a *trace* as a sequence in $\mathbf{Traces} \subseteq (\mathcal{L} \times A)^\omega$ representing the emission of an infinite execution of an MDP. $\mathbf{Paths}_{\text{fin}}$ (resp. $\mathbf{Traces}_{\text{fin}}$) is the set of finite prefixes of \mathbf{Paths} (resp. \mathbf{Traces}), as above. Finally, we define $|\rho|$ the length of a finite path ρ and (resp. $|o|$ the length of a finite trace o), i.e. the number of states (resp. labels) in the sequence.

Definition 2.3.3 (Scheduler). *A scheduler is a function $\sigma : \mathbf{Paths}_{\text{fin}} \rightarrow \mathcal{D}(A)$ that determines a distribution of actions to take based on the current path.*

This notion of scheduler encompasses well-studied classes of schedulers such as memoryless, deterministic, and randomised (*cf.* [7]).

2.3.3 Continuous-Time Markov Chains

All the models described so far are *discrete-time* models: they capture the order in which events occur, but not their timing. For instance, the MC from Example 2.3.1 represents naively the weather, but it cannot be used for weather forecasting since it gives no indication of the time elapsing between two weather conditions: it says that it will be cloudy then rainy, but it does not say when it is going to start raining.

Continuous-time Markov chains (CTMCs) are extensions of MCs modelling the order and the timing in which events occur.

Definition 2.3.4. *A CTMC is a tuple $\langle S, \mathcal{L}, \ell, R, \pi \rangle$ where S, \mathcal{L}, ℓ and π are defined as above, and $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is the transition rate function.*

The transition rate function assigns rates $r = R(s, s')$ to each pair of states $s, s' \in S$. A transition from s to s' can only occur if $R(s, s') > 0$. In this case, the probability of this transition to be triggered within $t \in \mathbb{R}_{>0}$ time-units is $1 - e^{-rt}$. When, from a state s , there are more than one outgoing transition with positive rate, we are in presence of a *race condition*. In this case, the first transition to be triggered determines the next state of the CTMC. According to these dynamics, the time spent in state s before any transition occurs, called *dwelt time*, is exponentially distributed with parameter $E(s) = \sum_{s' \in S} R(s, s')$, called *exit-rate* of s . The probability that the transition from s to s' is triggered is $R(s, s')/E(s)$ and is independent from the time at which it occurs.

Accordingly, for a CTMC \mathcal{M} , we construct the *embedded MC* of \mathcal{M} as $\text{emb}(\mathcal{M}) = \langle S, \mathcal{L}, \ell, \tau, \pi \rangle$ with transition probability distributions $\tau : S \rightarrow \mathcal{D}(S)$ are defined as

$$\tau(s)(s') = \begin{cases} R(s, s')/E(s) & \text{if } E(s) \neq 0 \\ 1 & \text{if } E(s) = 0 \text{ and } s = s' \\ 0 & \text{otherwise} \end{cases}$$

Remark 2.3.7 (Absorbing state). *A state s is called absorbing if $E(s) = 0$, that is, s has no outgoing transition. Accordingly, when the CTMC ends in an absorbing state it will remain in the same state indefinitely.*

Remark 2.3.8 (Initial state). *π may be a Dirac distribution concentrated in one state s . In such case, we can use $s_{init} = s$ instead of π to describe the initial distribution.*

Example 2.3.3. Let $\mathcal{M} = \langle S, \mathcal{L}, \ell, R, \pi \rangle$ be a CTMC where:

- $S = \{s_1, s_2, s_3\}$,
- $\mathcal{L} = \{\text{sunny, cloudy, rainy}\}$,
- $\ell(s_1) = \text{sunny}$, $\ell(s_2) = \text{cloudy}$ and $\ell(s_3) = \text{rainy}$,
- $\pi(s_1) = 1.0$ and $\pi(s_2) = \pi(s_3) = 0.0$, and
- $R(s_1, s_2) = 1.0$, $R(s_2, s_1) = 1.0$, $R(s_2, s_3) = 1.0$, and $R(s_3, s_2) = 1.5$.

\mathcal{M} could be depicted as in Figure 2.3.

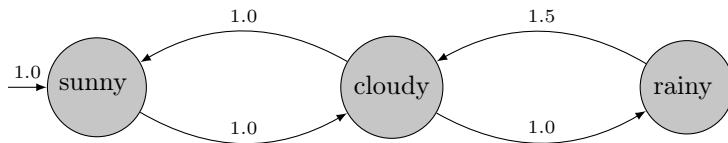


Figure 2.3 – A simple CTMC with three states

Paths and traces

As for MCs and MDPs, we introduce the notion of CTMC *paths* and *traces*. In contrast with MC and MDP paths, CTMC paths may contain dwell times. Here, for uniformity of treatment, the dwell times that are missing are denoted as \emptyset .

A CTMC path is a sequence in $\mathbf{Paths} \subseteq (S \times \mathbb{R}_{>0} \cup \{\emptyset\})^\omega$ representing an infinite execution of a CTMC.

A CTMC trace is a sequence in $\mathbf{Traces} \subseteq (\mathcal{L} \times \mathbb{R}_{>0} \cup \{\emptyset\})^\omega$ representing an infinite execution of a CTMC for which we cannot see the states. Finally, we define $\mathbf{Paths}_{\text{fin}}$ and $\mathbf{Traces}_{\text{fin}}$ respectively the set of prefixes of \mathbf{Paths} and \mathbf{Traces} .

We denote by $|\rho|$ (resp. by $|o|$) the length of a finite path ρ (resp. a finite trace $|o|$), i.e. the number of states in the sequence (resp. the number of labels).

Finally, we define

$$\mathcal{T}(o) = \{i \mid 0 \leq i < n, t_i \neq \emptyset\} \quad (2.2)$$

with $o = s_0 t_0 s_1 \dots s_n \in \mathbf{Traces}_{\text{fin}}$, i.e. the subset of indices of the trace o that correspond to actual dwell time measurement.

We define $\mathcal{T}(\rho)$ similarly, for ρ a finite path of length n .

2.4 Probability Measure of a Markov Model

The probability measure for discrete-time models (MCs and MDPs) must be defined differently for continuous-time models. Indeed, the probability measure for continuous-time models involves continuous probability distributions which must be handled differently than the discrete probability distributions used for discrete-time models.

2.4.1 Probability Measure of discrete-time models

Following the classical cylinder set construction [7, Ch10] we define the *cylinder* set of a finite path ρ as $cyl(\rho) = \{\hat{\rho} \in \mathbf{Paths} \mid \rho \in \text{pref}(\hat{\rho})\}$.

We define $\Sigma = \sigma(\{cyl(\rho) \mid \rho \in \mathbf{Paths}_{\text{fin}}\})$, the smallest σ -algebra that contains all the cylinder sets $cyl(\rho)$.

An MC $\mathcal{M} = \langle S, \mathcal{L}, \ell, \tau, \pi \rangle$ induces a probability space $(\mathbf{Paths}, \Sigma, Pr^{\mathcal{M}})$ where $Pr^{\mathcal{M}}$ denotes the (unique) probability measure such that for arbitrary $\rho = s_0 \dots s_n \in \mathbf{Paths}_{\text{fin}}$,

$$Pr^{\mathcal{M}}(cyl(\rho)) = \pi(s_0) \prod_{i=0}^{n-1} \tau(s_i)(s_{i+1}) := Pr^{\mathcal{M}}(\rho),$$

and an MDP \mathcal{M} induces a probability space $(\mathbf{Paths}, \Sigma, Pr^{\mathcal{M}})$ where $Pr^{\mathcal{M}}$ denotes the (unique) probability measure such that for arbitrary $\rho = s_0 a_0 \dots s_n \in \mathbf{Paths}_{\text{fin}}$,

$$Pr^{\mathcal{M}}(cyl(\rho)) = \pi(s_0) \prod_{i=0}^{n-1} \tau_{a_i}(s_i)(s_{i+1}) := Pr^{\mathcal{M}}(\rho).$$

For $\rho = s_0 s_1 \dots s_n$ a finite MC path and for $i \in \mathbb{N}_{\leq n}$, we define:

- $X_i: \mathbf{Paths}_{\text{fin}} \rightarrow S$ as $X_i(\rho) = s_i$,
- $Y_i: \mathbf{Paths}_{\text{fin}} \rightarrow \mathcal{L}$ as $Y_i(\rho) = \ell(s_i)$, and
- $O_i: \mathbf{Paths}_{\text{fin}} \rightarrow \mathbf{Traces}_{\text{fin}}$ as $O_i(\rho) = \ell(s_0)\ell(s_1)\dots\ell(s_i)$.

For $\rho = s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ an MDP path and for $i \in \mathbb{N}_{\leq n}$, we define:

- X_i and Y_i as above,
- $A_i: \mathbf{Paths}_{\text{fin}} \rightarrow A$ as $A_i(\rho) = a_i^{-1}$, and
- $O_i: \mathbf{Paths}_{\text{fin}} \rightarrow \mathbf{Traces}_{\text{fin}}$ as $O_i(\rho) = \ell(s_0)a_0\ell(s_1)a_1\dots\ell(s_{i-1})a_{i-1}\ell(s_i)$.

We call the *induced trace* of a finite path ρ the trace emitted by the execution represented by ρ , i.e. $O_{|\rho|}(\rho)$. For the sake of clarity, we sometimes simply write $O(\rho)$ instead of $O_{|\rho|}(\rho)$. However, given a finite trace o , several paths could describe the execution that has emitted o . We denote by $\mathbf{Paths}(o)$ the set of paths that induce o (for the sake of simplicity, we say that $\mathbf{Paths}(o)$ is the set of paths induced by o):

$$\mathbf{Paths}(o) = \{\rho \in \mathbf{Paths}_{\text{fin}} \mid O(\rho) = o\}.$$

1. A_i is defined for $i \in \mathbb{N}_{< n}$ since there is no action a_n

And we define the probability of a trace o as the sum of the probabilities of the paths in $\mathbf{Paths}(o)$:

$$Pr^{\mathcal{M}}(o) = \sum_{\rho \in \mathbf{Paths}(o)} Pr^{\mathcal{M}}(\rho)$$

A special case arises when the model is *deterministic*:

Definition 2.4.1 (Deterministic model). *A Markov model \mathcal{M} is deterministic if and only if*

$$\forall o \in \mathbf{Traces}_{fin}: |\{\rho \mid \rho \in \mathbf{Paths}(o) \wedge Pr^{\mathcal{M}}(\rho) > 0\}| \leq 1$$

Or equivalently: $Pr^{\mathcal{M}}(o) > 0 \implies \exists \rho \in \mathbf{Paths}(o) \text{ s.t. } Pr^{\mathcal{M}}(\rho) = Pr^{\mathcal{M}}(o)$

In practice if, for any state s and label ℓ of a Markov model \mathcal{M} , there exists at most one transition leaving s to a state labelled with ℓ , then \mathcal{M} is *deterministic*.

Example 2.4.1. Consider the non-deterministic MC depicted in Figure 2.4 where the labels are in red. For the sake of clarity, the probabilities have been omitted in the representation.

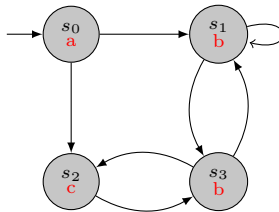


Figure 2.4 – A non-deterministic MC.

We can easily see that the induced trace of $s_0s_1s_3$ is $O_3(s_0s_1s_3) = abb$. Nevertheless, $\{s_0s_1s_3, s_0s_1s_1\} \subseteq \mathbf{Paths}(abb)$, and $Pr^{\mathcal{M}}(s_0s_1s_3) \cdot Pr^{\mathcal{M}}(s_0s_1s_1) > 0$. This proves that this model is non-deterministic. It's worth noting that the model becomes deterministic when the self-loop in s_1 is removed.

The likelihood of a finite path ρ under a model \mathcal{M} , denoted by $l(\mathcal{M}; \rho)$, is the probability that an infinite execution of \mathcal{M} starts by ρ :

$$l(\mathcal{M}; \rho) = Pr^{\mathcal{M}}(cyl(\rho)).$$

Similarly, the likelihood of a finite trace o under a model \mathcal{M} , denoted by $l(\mathcal{M}; o)$, is the probability that an infinite execution of \mathcal{M} starts by emitting o . This value is equal to

$$l(\mathcal{M}; o) = \sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{M}; \rho)$$

We extend the likelihood notion to sets of traces as follows: assuming \mathcal{O} is a finite set of traces drawn independently, the likelihood of \mathcal{O} under \mathcal{M} is

$$l(\mathcal{M}; \mathcal{O}) = \prod_{o \in \mathcal{O}} l(\mathcal{M}; o)$$

2.4.2 Probability Measure of continuous-time models

For $\rho = s_0 t_0 s_1 t_1 \dots s_n$ a finite CTMC path and for $i \in \mathbb{N}_{\leq n}$, we define:

- X_i and Y_i as above,
- $T_i: \mathbf{Paths}_{\text{fin}} \rightarrow \mathbb{R}_{>0} \cup \emptyset$ as $T_i(\rho) = t_i^2$, and
- $O_i: \mathbf{Paths}_{\text{fin}} \rightarrow \mathbf{Traces}_{\text{fin}}$ as $O_i(\rho) = \ell(s_0)t_0\ell(s_1)t_1 \dots \ell(s_{i-1})t_{i-1}\ell(s_i)$.

Since the probability that a CTMC stays in a state for an exact given dwell time is zero (i.e. the probability that a random variable following any continuous probability distribution is exactly equal to a given value is zero), the probability that a CTMC generates exactly any finite path is zero. Therefore, we cannot define the cylinder set of a CTMC finite path ρ as all finite paths starting with ρ , otherwise the probability of any path under any CTMC will always be zero. Instead, we define the cylinder set $\text{cyl}(s_0 t_0 \dots s_{n-1} t_{n-1} s_n)$ of a CTMC finite path as all infinite paths in $\rho \in \mathbf{Paths}$ such that $X_i(\rho) = s_i$ for $i = 0, \dots, n$ and $T_i(\rho) \leq t_i$ for $i \in \mathcal{T}(\rho)$ ³. $T_i(\rho)$ is not constrained for $0 \leq i < n$ such that $i \notin \mathcal{T}(\rho)$.

We define $\Sigma = \sigma(\{\text{cyl}(\rho) \mid \rho \in \mathbf{Paths}_{\text{fin}}\})$, the smallest σ -algebra that contains all the cylinder sets $\text{cyl}(\rho)$.

A CTMC \mathcal{M} induces a probability space $(\mathbf{Paths}, \Sigma, Pr^{\mathcal{M}})$ where $Pr^{\mathcal{M}}$ denotes the (unique) probability measure such that, for $\rho = s_0 t_0 \dots s_n \in \mathbf{Paths}_{\text{fin}}$,

$$Pr^{\mathcal{M}}(\text{cyl}(\rho)) = \pi(s_0) \prod_{i=0}^{n-1} \underbrace{\frac{R(s_i, s_{i+1})}{E(s_i)}}_{\text{from } s_i \text{ to } s_{i+1}} \cdot \prod_{i \in \mathcal{T}(\rho)} \underbrace{(1 - e^{-E(s_i) t_i})}_{\text{within } t_i \text{ time units}}.$$

We define the notion of *induced trace* for continuous-time models as for discrete-time models. For $o \in \mathbf{Traces}_{\text{fin}}$, the set of paths induced by o is

$$\mathbf{Paths}(o) = \{\rho \in \mathbf{Paths}_{\text{fin}} \mid O(\rho) = o\}$$

In contrast with discrete-time models, the likelihood of a finite path ρ is not equal to the probability of this finite path. Instead, the likelihood of ρ is given by

$$l(\mathcal{M}; \rho) = \pi(s_0) \prod_{i=0}^{n-1} \underbrace{\frac{R(s_i, s_{i+1})}{E(s_i)}}_{\text{from } s_i \text{ to } s_{i+1}} \cdot \prod_{i \in \mathcal{T}(\rho)} \underbrace{(E(s_i) e^{-E(s_i) t_i})}_{\text{in } t_i \text{ time units}}.$$

The likelihood of a finite trace o under a model \mathcal{M} , denoted by $l(\mathcal{M}; o)$, is the likelihood that an infinite execution of \mathcal{M} starts by any path inducing o :

$$l(\mathcal{M}; o) = \sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{M}; \rho)$$

Remark 2.4.1. If $\mathcal{T}(o) = \emptyset$, i.e. if the trace does not contain any dwell time, then $l(\mathcal{M}; o) = l(\text{emb}(\mathcal{M}); o)$

2. T_i is defined for $i \in \mathbb{N}_{< n}$
3. Reminder: \mathcal{T} is defined in (2.2)

We extend the likelihood notion to sets of traces as follow:
given \mathcal{O} a finite set of traces, the likelihood of \mathcal{O} under \mathcal{M} is

$$l(\mathcal{M}; \mathcal{O}) = \prod_{o \in \mathcal{O}} l(\mathcal{M}; o)$$

2.4.3 Measuring the difference between models

In this Subsection, we present the two measures used in this document to compare Markov models. This process is crucial as it allows for the evaluation of the performance of two Markov models, facilitating the selection of the most suitable one for a specific task. Additionally, comparing the output models of two learning methods in the same context enables the identification of the more efficient method for that context.

First, we introduce the notion of *trace equivance*:

Definition 2.4.2 (Trace equivance). *Two models \mathcal{M} and \mathcal{M}' are trace equivalent iff*

$$\forall o \in \mathbf{Traces}_{fin}^{\mathcal{M}} \cup \mathbf{Traces}_{fin}^{\mathcal{M}'} : l(\mathcal{M}; o) = l(\mathcal{M}'; o)$$

We write $\mathcal{M} \equiv \mathcal{M}'$.

Definition 2.4.3 (the loglikelihood distance). *Given two Markov models \mathcal{M} and \mathcal{M}' and a set of finite traces \mathcal{O} , the loglikelihood distance between \mathcal{M} and \mathcal{M}' is*

$$\mathbb{L}(\mathcal{M}, \mathcal{M}'; \mathcal{O}) = \frac{1}{|\mathcal{O}|} |\ln l(\mathcal{M}; \mathcal{O}) - \ln l(\mathcal{M}'; \mathcal{O})|$$

The *loglikelihood distance* is the average of the absolute differences between the loglikelihood under \mathcal{M} and \mathcal{M}' of \mathcal{O} .

Example 2.4.2. The first model is the MC depicted in Figure 2.1, and the two others are slightly modified versions of the first one.

Let $\mathcal{O} = \{\{\text{'sunny'}, \text{'cloudy'}, \text{'rainy'}, \text{'rainy'}\}, \{\text{'sunny'}, \text{'sunny'}, \text{'sunny'}, \text{'cloudy'}\}\}$. The loglikelihood of \mathcal{O} under each of these three model is $\ln l(\mathcal{M}_0; \mathcal{O}) = -2.816$, $\ln l(\mathcal{M}_1; \mathcal{O}) = -3.103$, $\ln l(\mathcal{M}_2; \mathcal{O}) = -2.469$. From this information, \mathcal{M}_2 is the most suitable for \mathcal{O} . In other words, if the point is to choose a model that maximises the probability of generating \mathcal{O} , one should choose \mathcal{M}_2 .

The loglikelihood distance on the other hand compares the likelihood of given set of traces under two models. Here, the loglikelihood distances are $\mathbb{L}(\mathcal{M}_0, \mathcal{M}_1; \mathcal{O}) = 0.288$, $\mathbb{L}(\mathcal{M}_0, \mathcal{M}_2; \mathcal{O}) = 0.347$. Therefore, if the objective is to choose the model the closer to \mathcal{M}_0 one should choose \mathcal{M}_1 .

2.5 Few words about Hidden Markov Models

2.5.1 The classic Hidden Markov Models

Initially introduced by Baum in the 1960s for speech recognition, Hidden Markov Models (HMMs) has now applications in various domains, such as bioinformatics[16,

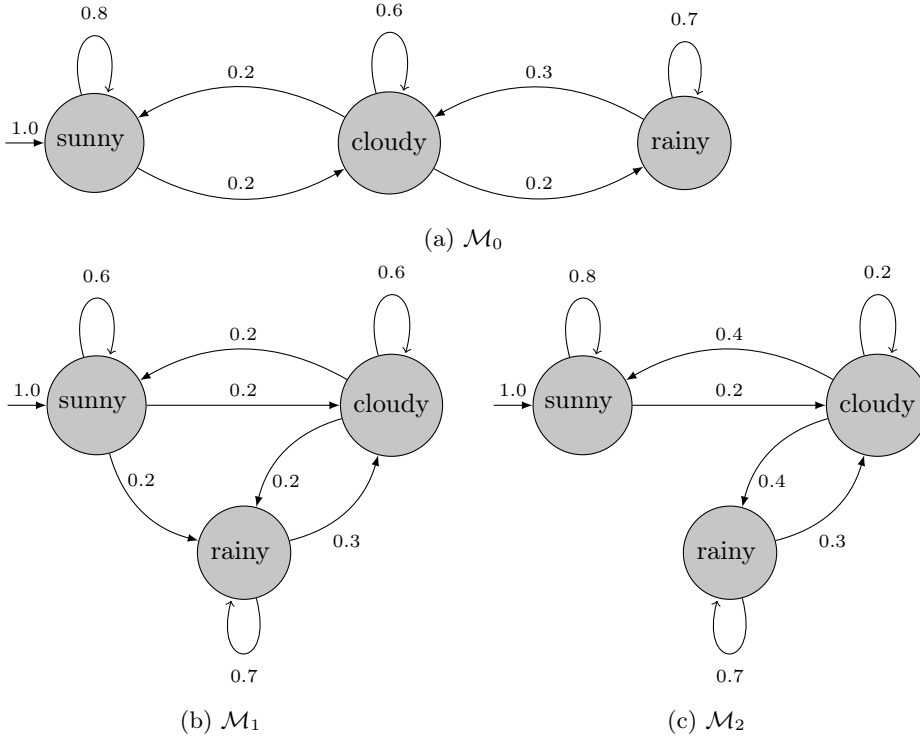


Figure 2.5 – Three slightly different MCs.

17, 18], computational finance [19] and cryptanalysis [20].

HMMs can be seen as extensions of MCs where each state is associated to a probability distribution over the set of labels \mathcal{L} . Formally:

Definition 2.5.1 (Hidden Markov Model). *An HMM is a tuple $\langle S, \mathcal{L}, a, b, \pi \rangle$ where S, \mathcal{L} and π are defined as above, and:*

1. $a : S \mapsto \mathcal{D}(S)$ is a transition function: the model moves from state s to s' with probability $a(s)(s')$, and
2. $b : S \mapsto \mathcal{D}(\mathcal{L})$ is the generation function: while in state s , the model generates ℓ with probability $b(s)(\ell)$.

Example 2.5.1. In this example, we build an HMM to estimate the current outdoor temperature (cold, hot, or temperate) based on the headwear (cap or hat) worn by individuals who enter a building. Such HMM could be $\mathcal{M} = \langle S, \mathcal{L}, a, b, \pi \rangle$ with

- $S = \{s_0, s_1, s_2\}$,
- $\mathcal{L} = \{\text{hat}, \text{cap}\}$,
- $a(s_0)(\text{hat}) = 0.8, a(s_0)(\text{cap}) = 0.2$
 $a(s_1)(\text{hat}) = 0.3, a(s_1)(\text{cap}) = 0.7$
 $a(s_2)(\text{cap}) = 1.0$.

- $b(s_0)(s_0) = 0.4, b(s_0)(s_1) = 0.6$
 $b(s_1)(s_0) = 0.3, b(s_1)(s_1) = 0.5, b(s_1)(s_2) = 0.2,$
 $b(s_2)(s_1) = 0.4, b(s_2)(s_2) = 0.6.$
- $\pi(s_0) = 1.0.$

According to the generation functions, we could see s_0 as a state representing the cold temperature, s_1 the tempered one and s_2 the warm one. However, these are interpretations: the only concrete observations are the labels ‘cap’ and ‘hat’. \mathcal{M} can be depicted as in Figure 2.6.

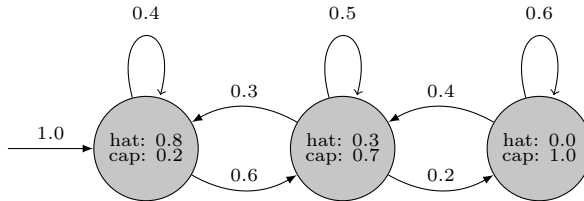


Figure 2.6 – An HMM for inferring outdoor temperature based on headwears.

Paths and traces HMM traces are defined as MC traces, while HMM paths contains labels as follow:

a path is a sequence in $\mathbf{Paths}^{\mathcal{M}} \subseteq (S \times \mathcal{L})^{\omega}$ representing an infinite execution of an HMM \mathcal{M} . We denote by $\mathbf{Paths}_{\text{fin}}^{\mathcal{M}} \subseteq (S \times \mathcal{L})^*$ the set of finite paths.

The likelihood of a finite path $\rho = s_0 \ell_0 \dots s_n \ell_n$ of length n under an HMM \mathcal{M} , denoted by $l(\mathcal{M}; \rho)$, is the probability that an infinite execution of \mathcal{M} starts by ρ :

$$l(\mathcal{M}; \rho) = Pr^{\mathcal{M}}(\text{cyl}(\rho)) = \pi(s_0) \cdot b(s_0)(\ell_0) \cdot \prod_{i=0}^{n-1} a(s_i)(s_{i+1}) \cdot b(s_{i+1})(\ell_{i+1}).$$

The likelihood of a finite trace $o = \ell_0 \dots \ell_n$ of length n under \mathcal{M} can be computed as follow:

$$l(\mathcal{M}; o) = \sum_{s_0, \dots, s_n \in S^n} l(\mathcal{M}; s_0 \ell_0 \dots s_n \ell_n)$$

HMMs differ from MCs only in the fact that HMM states are not associated with one label but a probability distribution over the labels. In the following, we present a method to construct, given any HMM, a trace equivalent MC.

Definition 2.5.2. *Given an HMM $\mathcal{M} = \langle S, \mathcal{L}, a, b, \pi \rangle$, the MC induced by \mathcal{M} is $\mathbb{M}(\mathcal{M}) = \langle S', \mathcal{L}', \ell, \tau, \pi' \rangle$ such that:*

1. $S' = \{(s, x) \mid \forall s \in S, x \in \mathcal{L}\},$
2. $\forall (s, x) \in S': \ell((s, x)) = x,$
3. $\forall (s, x) \in S': \pi'((s, x)) = \pi(s) \cdot b(s)(x),$ and
4. $\forall (s, x), (s', x') \in S': \tau((s, x))((s', x')) = a(s)(s') \cdot b(s')(x').$

Theorem 2.5.1. For all HMM \mathcal{M} :

$$\mathcal{M} \equiv \mathbb{M}(\mathcal{M})$$

Proof. Let $\mathcal{M} = \langle S, \mathcal{L}, a, b, \pi \rangle$ be an HMM and $o = \ell_0, \dots, \ell_n \in \mathcal{L}^n$, for any $n \in \mathbb{N}$, be any trace in $\mathbf{Traces}_{\text{fin}}^{\mathcal{M}}$ (and therefore in $\mathbf{Traces}_{\text{fin}}^{\mathbb{M}(\mathcal{M})}$). By definition:

$$\begin{aligned} l(\mathbb{M}(\mathcal{M}); o) &= \sum_{\rho \in \mathbf{Paths}(o)} l(\mathbb{M}(\mathcal{M}); \rho) \\ &= \sum_{s_0, \dots, s_n \in S^n} \pi'((s_0, \ell_0)) \cdot \prod_{i=0}^{n-1} \tau((s_i, \ell_i))((s_{i+1}, \ell_{i+1})) \\ &= \sum_{s_0, \dots, s_n \in S^n} \pi(s_0) \cdot b(s_0)(\ell_0) \cdot \prod_{i=0}^{n-1} a(s_i)(s_{i+1}) \cdot b(s_{i+1})(\ell_{i+1}) \\ &= l(\mathcal{M}; o) \end{aligned}$$

□

Example 2.5.2. Let \mathcal{M} be the HMM described in Example 2.5.1, $\mathbb{M}(\mathcal{M})$ is depicted in Figure 2.7.

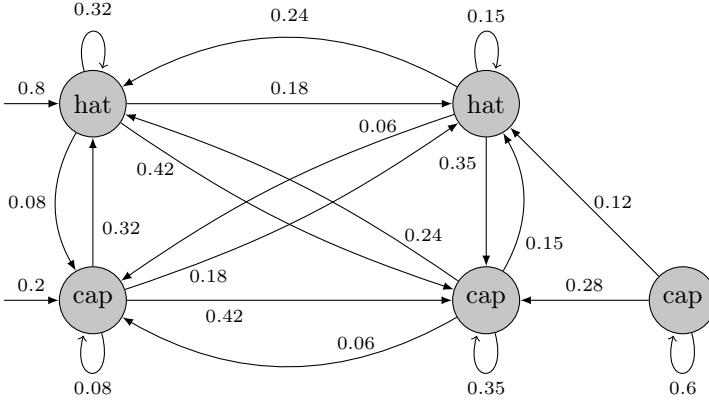


Figure 2.7 – MC induced by the HMM from Example 2.5.1.

In this thesis we do not consider HMMs but MCs instead, since there exists for any HMM a trace equivalent MC, by Theorem 2.5.1.

2.5.2 Gaussian Hidden Markov Models

In the previous Subsection we defined the classic notion of HMM, where the generation function is a discrete probability distribution over the set of labels \mathcal{L} . However, one could consider other probability distributions, even non discrete. In the same way, we could attach n probability distributions to each state: therefore,

at each transition step, n observations will be generated (one by each probability distribution attached to the current state).

A Gaussian observations Hidden Markov Model (GoHMM) is a HMM where each state is attached to n gaussian distributions instead of one discrete probability distribution over \mathcal{L} . Hence, a GoHMM generates n continuous observations at each transition step.

Definition 2.5.3 (Gaussian observations Hidden Markov Model). *A GoHMM is a tuple $\langle S, a, n, \{\theta_s\}_{s \in S}, \pi \rangle$ where S, a and π are defined as above, and:*

- n is the degree of the model,
- $\theta_s = \{\theta_s^{(1)}, \dots, \theta_s^{(n)}\}$ are the parameters used by the n Gaussian distributions to generate the observations while in state s , where $\theta_s^{(i)} = \{\mu_s^{(i)}, \sigma_s^{(i)}\}$.

In this context, an *observation* is a vector of n real values.

Remark 2.5.1. *In the literature, such model are sometimes called ‘independent GoHMM’, since all the n distributions are independent one from another.*

Paths and traces A GoHMM *path* is a alternating sequence of states and observations. Formally, a infinite path of a GoHMM \mathcal{M} is a sequence in $\mathbf{Paths}^{\mathcal{M}} \subseteq (S \times \mathbb{R}^n)^\omega$, and a finite path is a sequence in $\mathbf{Paths}_{\text{fin}}^{\mathcal{M}} \subseteq (S \times \mathbb{R}^n)^*$. A GoHMM *trace* is a path without the states, i.e. is a sequence of observations. A trace of \mathcal{M} is a sequence in $\mathbf{Traces}^{\mathcal{M}} \subseteq (\mathbb{R}^n)^\omega$, and a finite trace is a sequence in $\mathbf{Traces}_{\text{fin}}^{\mathcal{M}} \subseteq (\mathbb{R}^n)^*$.

Part I

State of the Art

Foreword

This section provides an overview of the current state-of-the-art techniques to learn Markov models. The main goal of this section is to provide the reader with valuable insights into the field's present advancements and develop a deeper appreciation for the improvements introduced in this thesis. It is beyond the scope of this section to provide technical details of each of the presented learning techniques. The interested reader may find such details by following the references given throughout the section.

Chapter 3

Methods

In this chapter we present the main algorithms for learning stochastic models with their strengths and weaknesses. These algorithms solve the problem of learning stochastic models using different approaches.

Maximum Likelihood Estimation (MLE). MLE works under the assumption that the model to learn belongs to a parametric family of models. The goal of maximum likelihood estimation is to determine the parameter values for which the observed data achieve the highest joint likelihood. Typically, the joint likelihood surface is not convex, and computing its global maximum is often computationally intractable. The literature presented a number of techniques to approximate the MLE problem: Expectation-Maximisation (EM) algorithm [21], Minorize-Maximization (MM) algorithm [22, 23], Monte Carlo EM algorithm [24]. One of the most popular is the Expectation-Maximisation method.

The EM algorithm is an iterative optimisation technique used for problems involving latent variables. It operates by iteratively improving parameter estimates in situations where data is incomplete or has hidden variables. The algorithm has two main steps in each iteration: the E-step (Expectation step) and the M-step (Maximisation step). In the E-step, it computes the expected values of the latent variables based on the current parameter estimates. In the M-step, it updates the parameter estimates by maximising the likelihood of the observed data, treating the expected latent variable values from the E-step as if they were observed. These steps alternate until convergence, with each iteration ideally bringing the parameter estimates closer to the true underlying values. EM aims to maximise the likelihood of the observed data (observed variables) while accounting for unobserved variables (*latent variables*).

The Baum-Welch (BW) algorithm [25, 26, 27] is a specific application of the EM algorithm to the context of Markov models. In this context, the E-step involves computing the probabilities of hidden states given observations, while the M-step updates the model's parameters based on these probabilities. Therefore, the likelihood of the training set under the hypothesis increases all along the learning process. Despite its sensitivity to initial probabilities and the possibility of being

trapped in local optima, this technique has proven to be successful in practical applications on numerous occasions [26, 28, 29, 30].

Recently, moment-based approaches a.k.a. *spectral learning* have gained popularity as an alternative approach to EM [31, 32].

State-Merging. The second idea is to start with a model possessing a large number of states to precisely represent the training set. Subsequently, a series of iterations are employed to systematically merge the states of this automaton, thereby enhancing the model’s conciseness and compactness. This time, the likelihood of the training set under the hypothesis decreases all along the learning process.

Learning with queries. Rather than learning from a training set alone, the third idea involves interacting with the system under learning (SUL). Thus, it is not necessary to have a training set, but the algorithm must be able to interact with the SUL.

These algorithms are inspired by the L^* algorithm, developed by Angluin in 1987 [33], to learn regular languages. The L^* algorithm operates within the framework of a ‘teacher’ and a ‘learner.’ The ‘teacher’ represents the system or process whose underlying formal language needs to be inferred. The learner, on the other hand, is the algorithm itself, seeking to learn the language through interactions with the teacher. The teacher provides responses to membership queries and equivalence queries posed by the learner. Membership queries involve the learner asking whether specific strings belong to the language, and the teacher responds with a ‘yes’ or ‘no’. Equivalence queries are more comprehensive; the learner proposes a hypothesis language, and the teacher responds with a counterexample that highlights any discrepancies between the hypothesis and the actual language. By iteratively refining its hypothesis through a combination of membership and equivalence queries, the learner endeavours to converge towards an accurate description of the language underlying the system, effectively bridging the gap between observed behaviour and the formal language that governs it.

Bayesian learning. Finally, the last approach encompasses algorithms using Bayesian methods. This category contains an algorithm based on Bayesian inference to learn MCs [34] and an algorithm using Bayesian Reinforcement learning to learn MDPs [35].

These methods belong to two categories, active and passive. *Active* learning methods learn from interactions with the SUL, while *passive* methods learn from the training set only. Active learning methods are usually more efficient (in terms of data), but can be used only if it is possible to interact with the SUL.

Some learning methods allow the user to decide the size (i.e. the number of states) of the output model, preventing the algorithm from generating models too large to be efficiently analysed. The downside of such a feature consists in the fact that, if the number of states requested is too large (resp. small), the output model may overfit (resp. underfit) the training set.

Some learning methods described below assume the Markov model underlying the SUL to be deterministic. When such methods are exercised with a SUL that is non-deterministic, they are not granted to converge to the true model, instead, they will return a deterministic model that approximates the SUL. Typically, the approximated model is larger than the SUL.

This chapter is divided in three sections corresponding to the first three different approaches, the last not being covered in this thesis.

3.1 The Expectation-Maximisation approach

Expectation-Maximisation (EM) is a method to estimate a mixture of parameters. Although the EM algorithm had been used in various contexts since the 1950s [36, 37], it was not formally defined until 1977 by Dempster, Laird, and Rubin [21].

The EM algorithm assumes that the training set is incomplete: the goal of the EM algorithm is to maximise the likelihood of the parameters of a mixture model assuming that some data is missing in the available training set. The EM algorithm converges to a local maximum of the likelihood of the training set [38].

The EM algorithm is typically applied to solve the maximum likelihood estimation problem for a set of parameters θ and a set of observed data X , when the likelihood function $l(\theta; X)$ is more naturally expressed as the marginal likelihood relative to a set of unobserved *latent data* (or missing values) Z , i.e.,

$$l(\theta; X) = \sum_z Pr^\theta(z) Pr^\theta(X | z) \quad (3.1)$$

The above likelihood function is often intractable since z is unobserved and its distribution $Pr^\theta(z)$ is typically determined by the parameters θ .

The EM algorithm is an iterative optimisation technique aimed at maximising Equation (3.1). Starting from an initial parameter estimate θ_0 , the current parameter estimate θ_m is updated by applying the following steps:

Expectation step (E-step) Compute the expected value of the log-likelihood function relative to the conditional distribution of Z , given the observed data X and the current parameter value estimates θ_m

$$Q(\theta|\theta_m) = \sum_z Pr^{\theta_m}(z | X) \ln Pr^\theta(X, z) \quad (3.2)$$

Maximisation step (M-step) The next parameter estimates θ_{m+1} are found as those achieving the maximum value of $Q(\theta|\theta_m)$, called the *surrogate function*. Formally

$$\theta_{m+1} = \arg \max_{\theta} Q(\theta|\theta_m).$$

Notably, the new parameter estimate improves with respect to the previous one in the sense that $l(\theta_m; X) \leq l(\theta_{m+1}; X)$.

Theorem 3.1.1 (Convergence of the EM algorithm). *The EM algorithm converges to a local maximum of the likelihood of the training set [38].*

For the proof, see Appendix A.1.

The EM algorithm, while a powerful tool for solving problems involving latent variables, can exhibit slow convergence to local optima. The method’s performance is contingent on the initial choice of parameter values: some initial parameter estimates may lead the procedure to converge to local maxima rather than global ones. Consequently, selecting appropriate initial parameter values becomes a critical consideration for enhancing both the efficiency and effectiveness of the optimisation procedure. This phenomenon has been extensively studied in the literature [38, 39]. To mitigate the impact of slow convergence and improve the algorithm’s performance, practitioners often employ strategies like multiple restarts with varying initial values and leveraging domain knowledge to provide more informed initial estimates.

The Baum-Welch algorithm is an application of the Expectation Maximisation optimisation framework to Markov models. Initially designed in the early 1970s by Baum and Welch for HMM learning [26, 27], the BW algorithm has been extended to various families of Markov models. In this context, the observed data X (i.e. the training set) is a set of traces, while the latent data Z is the corresponding set of paths. The parameters to estimate, θ , are the transition probabilities, or transition rates for CTMCs.

As EM, BW assumes that the model belongs to a parametric family of models. Here, the family is defined by the type of Markov model (e.g. MC, MDP, etc . . .) and the number of states. Therefore, BW allows the user to decide the size of the output model. As the EM algorithm, BW is a passive learning method. The statistical guarantees of the BW algorithm convergence are discussed in [40].

Finally, BW is known to be costly in terms of time and memory complexity: in [41], the authors cite several cases where, due to its cost, the BW algorithm could not be applied. Several attempts have been made to improve BW efficiency [42, 43, 44].

Baum-Welch in a nutshell.

In the context of learning a Markov model from a set of traces \mathcal{O} , the parameters to estimate are the transition probabilities/rates and the initial state probability distribution π , the set of observed data are the traces in \mathcal{O} , and the latent data are the paths. Therefore, the surrogate function Q becomes:

$$Q(\mathcal{M} | \mathcal{M}_m) = \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{M}_m; \rho) \ln [l(\mathcal{M}; \rho)]$$

The Baum-Welch algorithm starts with an initial hypothesis \mathcal{M}_0 which is iteratively updated, following the EM algorithm, in a way that the likelihood is nondecreasing at each step, that is $l(\mathcal{M}_m; \mathcal{O}) \leq l(\mathcal{M}_{m+1}; \mathcal{O})$, until the likelihood

```

BW( $\mathcal{O}, \mathcal{M}_0$ )
1   $i = 0$ 
2  repeat
3       $(\alpha, \beta) = \text{FORWARD-BACKWARD}(\mathcal{M}_i, \mathcal{O})$  // E-step
4       $\mathcal{M}_{i+1} = \text{UPDATE}(\mathcal{M}_i, \mathcal{O}, \alpha, \beta)$  // M-step
5       $i = i + 1$ 
6  until  $l(\mathcal{M}_i; \mathcal{O}) - l(\mathcal{M}_{i-1}; \mathcal{O}) \leq \epsilon$ 
7  return  $\mathcal{M}_i$ 

```

Algorithm 3.1.1 – Baum-Welch algorithm

difference between the current and the previous hypothesis goes below a fixed threshold ϵ (*cf.* Algorithm 3.1.1).

As the UPDATE procedure does not introduce or eliminate any states, the resulting model will contain the same number of states as the initial hypothesis. Consequently, by providing the initial hypothesis, the user has the freedom to determine the size of the output model.

The UPDATE procedures for MCs, MDPs and CTMCs are described in the following Subsections. The justifications are given in Appendix A.2, A.3 and A.4.

3.1.1 The Baum-Welch algorithm for learning MCs

In this Subsection we describe the UPDATE procedure for the BW algorithm applied to MC learning. For the justification, see Appendix A.2.

For $o = \ell_0 \dots \ell_T$ a trace and an MC \mathcal{M} , we define the forward and the backward functions $\alpha_o, \beta_o: S \times \{0 \dots T\} \rightarrow [0, 1]$ as

$$\begin{aligned} \alpha_o(s, i) &= Pr^{\mathcal{M}}[Y_{0:i} = \ell_0 \dots \ell_i, X_i = s], \text{ and} \\ \beta_o(s, i) &= Pr^{\mathcal{M}}[Y_{i:T} = \ell_i \dots \ell_T | X_i = s]. \end{aligned}$$

These can be calculated according to the following recurrences

$$\begin{aligned} \alpha_o(s, i) &= \begin{cases} \llbracket \ell_0 = \ell(s) \rrbracket \cdot \pi(s) & \text{if } i = 0 \\ \llbracket \ell_i = \ell(s) \rrbracket \cdot \sum_{s' \in S} \alpha_o(s', i-1) \cdot \tau(s')(s) & \text{if } 0 < i \leq T \end{cases} \\ \beta_o(s, i) &= \begin{cases} \llbracket \ell_T = \ell(s) \rrbracket & \text{if } i = T \\ \llbracket \ell_i = \ell(s) \rrbracket \cdot \sum_{s' \in S} \tau(s)(s') \cdot \beta_o(s', i+1) & \text{if } 0 \leq i < T \end{cases} \end{aligned}$$

Additionally we define, for o a trace of length T , the two functions $\gamma_o: S \times \{0 \dots T\} \rightarrow$

$[0, 1]$ and $\xi_o: S \times \{0 \dots T-1\} \times S \rightarrow [0, 1]$ as

$$\gamma_o(s, i) = Pr^{\mathcal{M}}[X_i = s | O_T = o], \quad (3.3)$$

$$\xi_o(s, i)(s') = Pr^{\mathcal{M}}[X_i = s, X_{i+1} = s' | O_T = o]. \quad (3.4)$$

Intuitively, $\gamma_o(s, i)$ is the likelihood of being in state s at the i -th steps given that the trace o has been observed, and $\xi_o(s, i)(s')$ is the likelihood that the i -th transition is the transition from s to s' given that the trace o has been observed. Note that these definitions hold for MCs as well as for any kind of Markov model.

Thus:

$$\gamma_o(s, i) = \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

$$\xi_o(s, i)(s') = \frac{\alpha_o(s, i) \cdot \tau(s)(s') \cdot \beta_o(s', i+1)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

For MCs, the UPDATE procedure updates π and τ as follows:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{u \in S} \gamma_o(u, 0)}$$

$$\hat{\tau}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i)}$$

Remark 3.1.1. *These values for $\hat{\pi}(s)$ and $\hat{\tau}(s)(s')$ are simply the analytic solution of $\arg \max_{\mathcal{M}} Q(\mathcal{M} | \mathcal{M}_m)$. The justifications are given in the Appendix A.2.*

Remark 3.1.2. *One may incur in the situation where $\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i) = 0$, indicating that the state s does not play a role in the observed dynamics. In this case the update procedure leaves the distribution $\tau(s)$ unchanged.*

3.1.2 The Baum-Welch algorithm for learning MDPs

In this Subsection we describe the UPDATE procedure for the BW algorithm applied to MDP learning. For the justification, see Appendix A.3.

For $o = \ell_0 a_0 \dots \ell_T$ a trace and an MDP \mathcal{M} , we define the forward and the backward functions $\alpha_o, \beta_o: S \times \{0 \dots T\} \rightarrow [0, 1]$ as

$$\alpha_o(s, i) = Pr^{\mathcal{M}}[Y_{0:i} = \ell_0 \dots \ell_i, X_i = s | A_{0:i-1} = a_0 \dots a_{i-1}], \text{ and}$$

$$\beta_o(s, i) = Pr^{\mathcal{M}}[Y_{i:T} = \ell_i \dots \ell_T | A_{i:T-1} = a_i \dots a_{T-1}, X_i = s].$$

These can be calculated according to the following recurrences

$$\alpha_o(s, i) = \begin{cases} \llbracket \ell_0 = \ell(s) \rrbracket \cdot \pi(s) & \text{if } i = 0 \\ \llbracket \ell_i = \ell(s) \rrbracket \cdot \sum_{s' \in S} \alpha_o(s', i-1) \cdot \tau_{a_{i-1}}(s')(s) & \text{if } 0 < i \leq T \end{cases} \quad (3.5)$$

$$\beta_o(s, i) = \begin{cases} \llbracket \ell_T = \ell(s) \rrbracket & \text{if } i = T \\ \llbracket \ell_i = \ell(s) \rrbracket \cdot \sum_{s' \in S} \tau_{a_i}(s)(s') \cdot \beta_o(s', i+1) & \text{if } 0 \leq i < T \end{cases} \quad (3.6)$$

Thus:

$$\gamma_o(s, i) = \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

$$\xi_o(s, i)(s') = \frac{\alpha_o(s, i) \cdot \tau_{a_i}(s)(s') \cdot \beta_o(s', i+1)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

For MDPs, the UPDATE procedure updates π and τ as follows:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{u \in S} \gamma_o(u, 0)}$$

$$\hat{\tau}_a(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \xi_o(s, i)(s') \cdot \llbracket a_i = a \rrbracket}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i) \cdot \llbracket a_i = a \rrbracket}$$

with γ and ξ defined as in (3.3) and (3.4).

Remark 3.1.3. *As for MCs, one may incur in the situation where $\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i) \cdot \llbracket a_i = a \rrbracket = 0$, indicating that the state s does not play a role in the observed dynamics. In this case the update procedure leaves the distribution $\tau_a(s)$ unchanged.*

3.1.3 The Baum-Welch algorithm for learning CTMCs

Let $\mathcal{M} = \langle S, \mathcal{L}, \ell, R, \pi \rangle$ be a CTMC. For all states $s, s' \in S$, we define $\tau(s)(s') = R(s)(s')/E(s)$, and $\lambda_s = E(s)$. Intuitively, when \mathcal{M} is in state s , it waits for a duration exponentially distributed with parameter λ_s and moves to state s' with probability $\tau(s)(s') := \tau_{s, s'}$.

In the following we show how the Baum-Welch UPDATE procedure works for CTMCs. For the justification, see Appendix A.4.

For $o = \ell_0 t_0 \dots \ell_T$ a timed trace and a CTMC \mathcal{M} , we define the forward and the backward functions $\alpha_o, \beta_o: S \times \{0..T\} \rightarrow [0, 1]$ as

$$\alpha_o(s, i) = Pr^{\mathcal{M}}[Y_{0:i} = \ell_0 \dots \ell_i, X_i = s | T_{0:i-1} = t_0 \dots t_{i-1}], \text{ and}$$

$$\beta_o(s, i) = Pr^{\mathcal{M}}[Y_{i:T} = \ell_i \dots \ell_T | T_{i:T-1} = t_i \dots t_{T-1}, X_i = s].$$

These can be calculated according to the following recurrences

$$\alpha_o(s, i) = \begin{cases} \llbracket \ell_0 = \ell(s) \rrbracket \cdot \pi(s) & \text{if } i = 0 \\ \llbracket \ell_i = \ell(s) \rrbracket \cdot \sum_{s' \in S} \alpha_o(s', i-1) \cdot \tau(s')(s) \cdot \lambda_{s'} e^{-\lambda_{s'} t_i} & \text{if } 0 < i \leq T \end{cases}$$

$$\beta_o(s, i) = \begin{cases} \llbracket \ell_T = \ell(s) \rrbracket & \text{if } i = T \\ \llbracket \ell_i = \ell(s) \rrbracket \cdot \sum_{s' \in S} \tau(s)(s') \cdot \lambda_s e^{-\lambda_s t_i} \cdot \beta_o(s', i+1) & \text{if } 0 \leq i < T \end{cases}$$

Thus:

$$\gamma_o(s, i) = \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

$$\xi_o(s, i)(s') = \frac{\alpha_o(s, i) \cdot \lambda_s e^{-\lambda_s t_i} \cdot \tau(s)(s') \cdot \beta_o(s', i+1)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

Given a set of timed traces \mathcal{O} , the UPDATE procedure updates π and R as follows:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{u \in S} \gamma_o(u, 0)}$$

$$\hat{R}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i \gamma_o(s, i)}$$

with γ and ξ defined as in (3.3) and (3.4).

Remark 3.1.4. *As for MCs and MDPs, one may incur in the situation where $\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|-1} \gamma_o(s, i) = 0$, indicating that the state s does not play a role in the observed dynamics. In this case the update procedure leaves the distribution $\tau(s)$ unchanged.*

Few words about the initial hypothesis. Usually, BW starts with a hypothesis where all the transitions are randomly initialised such that all of them have a non-zero probability. As a result, the original hypothesis exhibits sufficient expressiveness, as the likelihood of any trace in the training set under the hypothesis surpasses zero. Therefore, we do not encounter situations where the likelihood of a trace in the training set under the current hypothesis is zero, as in Remark 3.1.2, 3.1.3 and 3.1.4. However, since the quality of the output model depends on the initial hypothesis (the local optimum reached at the end of a BW execution depends on the initial hypothesis and the stop condition), the issue of obtaining a high-quality initial hypothesis while minimising costs has been thoroughly investigated in [45]. So far, the commonly used technique is *random restart*: the learning process is repeated n times (with n being an arbitrary value) using n different random initial hypotheses, and only the best output model in terms of loglikelihood distance under a test set is retained. This method is the simplest one and performs reasonably well, but it is not very efficient in terms of running time.

3.2 The State-Merging approach

The state-merging approach is a method commonly used in Markov model learning algorithms. Starting from a maximal tree-shaped model, this approach iteratively merges pairs of states based on their similarity until no further state aggregations are possible (see Figure 3.1). The maximal tree-shaped model is a *prefix tree acceptor* or one of its extension.

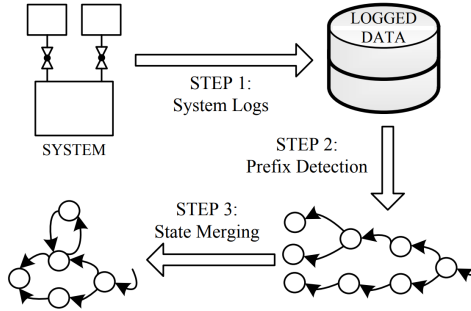


Figure 3.1 – State merging approach workflow [2]

Definition 3.2.1 (Prefix Tree Acceptor). A *Prefix Tree Acceptor (PTA)* is a tree-shaped Markov chain built over a set of traces \mathcal{O} . Each state of the PTA corresponds to a prefix in $\text{pref}(\mathcal{O})$ and is labelled with the last symbol of its corresponding prefix. For any $w \in \text{pref}(\mathcal{O})$, there exists exactly one path ρ in the PTA build over \mathcal{O} such that

$$O(\rho) = w \wedge Pr(\rho) > 0.$$

The probability of the transition from s , corresponding to the prefix w , to s' is equal to

$$\frac{|\{o \in \mathcal{O} | w \cdot \ell(s') \in \text{pref}(o)\}|}{|\{o \in \mathcal{O} | w \in \text{pref}(o) \wedge |o| > |w|\}|}.$$

Example 3.2.1. The PTA for a set of words containing ‘aa’ twice, ‘aaaa’ five times, ‘ab’ and ‘abaa’ once, ‘abaaa’ three times, ‘abab’ twice and ‘ababa’ 8 times is depicted in Figure 3.2.

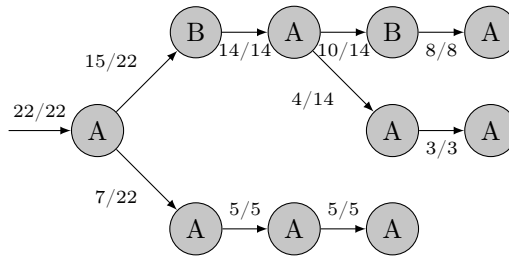


Figure 3.2 – Example PTA

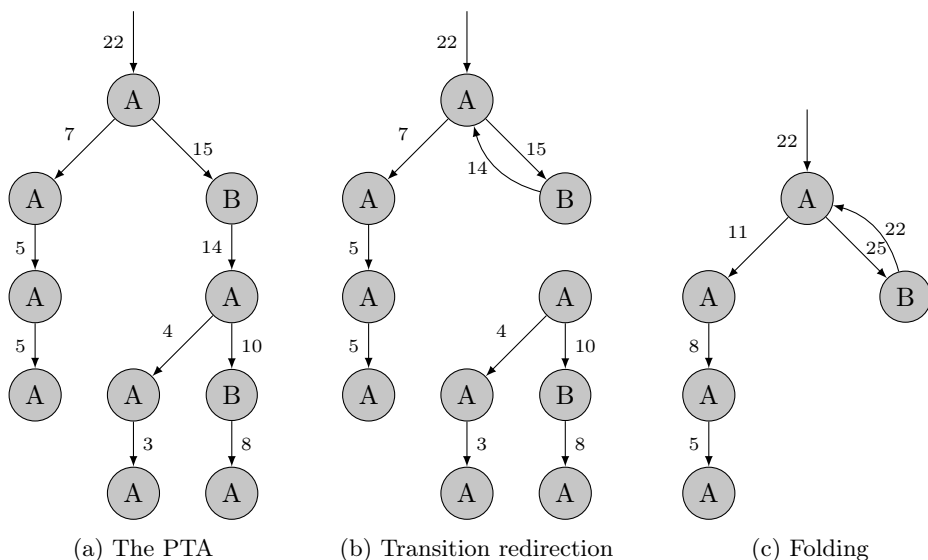


Figure 3.3 – Merge of s_{ABA} into s_A , where only the numerators of the transition probabilities are shown.

The seminal algorithm based on this approach is Alergia, designed by Carrasco and Oncina in the 1990s [46, 47], to learn MCs. Alergia has been adapted to CTMCs by Sen et al. in 2004 [48] and to MDPs by Mao et al. in 2012 [49]. The same year, Chen et al. developed an active extension of the latter [50].

In 2000, Thollard et al. proposed another state-merging based algorithm to learn MCs: the MDI algorithm (for Minimal Divergence Inference) [51].

We present these five algorithms in the five following Subsections.

3.2.1 Alergia: a learning algorithm for MCs

Given a training set \mathcal{O} and a confidence level parameter α , Alergia starts by building the *prefix tree acceptor* (PTA) for \mathcal{O} (see example 3.2.1). Then, the algorithm checks the compatibility of each pair of states in a top-down order (starting from the root state and progressing towards the leaf states) using the Hoeffding bound [52] with the parameter α , and merges each compatible pair (see example 3.3). It stops when no pair of states can be merged.

Due to its exclusive reliance on the training set for learning, Alergia is a passive learning algorithm.

It's easy to see that when using Alergia, the user cannot decide a priori what size the output model will be. Indeed, it is impossible to know in advance how many times the algorithm will perform a merging operation during its execution.

Furthermore, the model resulting from an Alergia run is necessarily deterministic. Indeed, Alergia starts with a PTA that is deterministic by construction and will

then only merge compatible states which have the same label. As a consequence, state merging cannot introduce non-determinism. When employed on a dataset that has been generated by a non-deterministic model, Alergia produces a deterministic model that approximates the original one. In this case, the learned model tends to be much larger than the original.

The Alergia algorithm is guaranteed to converge to a deterministic MC that approximates the underlying probabilistic process as more sequences are provided [46].

Finally, the time complexity of the Alergia algorithm is relatively efficient. The algorithm's complexity is polynomial in the size of the input sequences.

An empirical comparison of Alergia and BW on two small MCs is proposed in Section 7.5.1.

3.2.2 TAlergia: a learning algorithm for CTMCs

In the following we describe the Alergia extension for CTMC learning presented in [48]. No name have been assigned to this algorithm in [48] neither in the literature. In this document, we call it TAlergia, for Timed-Alergia.

TAlergia differs from Alergia on two points: the PTAs and the compatibility tests used by TAlergia are slightly adapted to include the dwell times. An example of TPTA is given in Figure 3.4.

In TAlergia, the training set \mathcal{O} contains sequences of label-dwell time pairs. These dwell time must be included in the initial prefix tree. To do so, TAlergia uses *timed prefix tree acceptors* (TPTAs), that are PTAs where each state is also associated to an average empirical dwell time \hat{t}_s , i.e the inverse of the empirical exit rate.

Two TPTA states are compatible if the transition probability distributions over their successor states as well as their exit-rates are compatible. As Alergia, TAler-

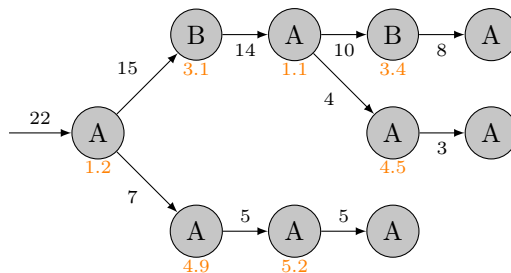


Figure 3.4 – An example of TPTA, with empirical dwell times in orange, where only the numerators of the transition probabilities are shown.

gia uses the Hoeffding bound to check the compatibility of the transition probability distributions. Additionally, TAlergia checks the exit-rates compatibility using an F -test [53].

Apart from the two previous points, TAlergia is identical to Alergia. Therefore, as Alergia, TAlergia is a passive learning algorithm, doesn't allow the user to decide the size of the output model, and cannot learn non-deterministic models.

As Alergia, TAlergia is guaranteed to converge to a deterministic CTMC, as the training set grows. The proof in [48] follows the same structure as the one for Alergia in [46].

3.2.3 IOAlergia: a learning algorithm for MDPs

IOAlergia [49] differs from Alergia on two points: the PTAs and the compatibility tests used by IOAlergia are slightly adapted to include MDP actions.

In IOAlergia, the training set \mathcal{O} contains sequences of label-action pairs. These actions must be included in the initial prefix tree. To do so, IOAlergia uses IOP-TAs, that are PTAs where each edge is associated to an action. An example of IOPTA is given in Figure 3.5.

IOAlergia compatibility checks are therefore adapted to include these actions.

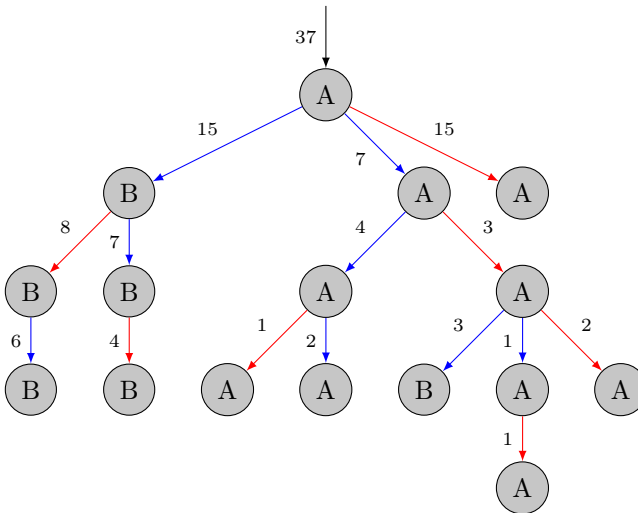


Figure 3.5 – An example of IOPTA with two actions ‘red’ and ‘blue’, where only the numerators of the transition probabilities are shown.

Apart from the two previous points, IOAlergia is identical to Alergia. Therefore, as Alergia, IOAlergia is a passive learning algorithm, doesn't allow the user to decide the size of the output model, and cannot learn non-deterministic models.

The proof of convergence for IOAlergia in [49] follows the same structure as the one for TAlergia [48] and for Alergia [46].

Due to the introduction of actions, learning an MDP is more complex than learning a MC or a CTMC. Indeed, the number of transition probability/rate to estimate while learning a MC/CTMC is equal to $|S|^2$, and is equal to $|S|^2 \times |A|$ while learning an MDP. Hence, when considering the identical number of states within the underlying Markov model of the SUL, achieving a comparable level of output quality in loglikelihood distance necessitates a larger training set for learning an MDP in contrast to MC or CTMC.

3.2.4 MDI: a learning algorithm for MCs

Alergia first evaluates if a pair of states are 'close enough' and then, if they are, merges these two states. On the contrary, MDI starts by merging two states and then compares if the resulting model is 'close enough' to the previous one (before merging) [51]. If it is not the case, the merge is discarded.

MDI uses the KL divergence (see [54]) to compare two models. Let $\mathcal{M}^{(0)}$ be the PTA build from the training set, $\mathcal{M}^{(n)}$ be the temporary model obtained after merging two states at any moment during the MDI execution, and $\mathcal{M}^{(n+1)}$ be another model resulting of the merge of two states in $\mathcal{M}^{(n)}$. $\mathcal{M}^{(n)}$ is kept only if the KL divergence increase caused by the merge leading to $\mathcal{M}^{(n+1)}$ is small enough relatively to the size reduction. Formally, $\mathcal{M}^{(n)}$ is kept if

$$\frac{\mathbb{K}(\mathcal{M}^{(0)}, \mathcal{M}^{(n+1)}; \mathcal{O}) - \mathbb{K}(\mathcal{M}^{(0)}, \mathcal{M}^{(n)}; \mathcal{O})}{|\mathcal{M}^{(n)}| - |\mathcal{M}^{(n+1)}|} < \epsilon,$$

with \mathcal{O} the training set, and ϵ a given compatibility threshold.

In [51], the authors show empirically that MDI outperforms Alergia in terms of *perplexity* (capability to guess the next symbol) and generalisation. On the other hand, MDI incurs higher computational demands compared to some other algorithms, as it necessitates the calculation of two KL divergences for each state merge.

3.3 Learning with Queries

In 1987, D. Angluin developed an algorithm capable of learning a language using a DFA (deterministic finite automaton) as a model, through interactions with a 'teacher' [33]. The interactions between the algorithm and the teacher involve two types of questions:

1. Membership queries: The algorithm asks the teacher whether a given word belongs to the language it needs to learn, and the teacher responds with either a yes or a no.
2. Equivalence queries: The algorithm asks the teacher whether the DFA it has constructed corresponds to the language it was supposed to learn. Here, the teacher responds with either a yes or with a counterexample, which is a word that is accepted by the DFA but not the language being learned, or vice versa.

The algorithm, called L^* , uses the answers to these questions to iteratively construct a DFA that represents the language it is trying to learn. By using these queries, L^* can learn a language efficiently, without having to examine all possible words in the language.

In practice, such a teacher does not exist: instead, the algorithm accesses the SUL which acts as a black box. The difficulty of the implementation of such a learning method lies on the one hand in the algorithm itself and on the other hand in the development of methods to efficiently simulate the teacher from the black box.

Angluin's L^* algorithm became rapidly the foundation for many active automata learning algorithms to learn other formalisms, such as Mealy machines [55, 56] and extended finite state-machines [57], non-deterministic Mealy machines [58], and MDPs (called L_{MDP}^*) [59].

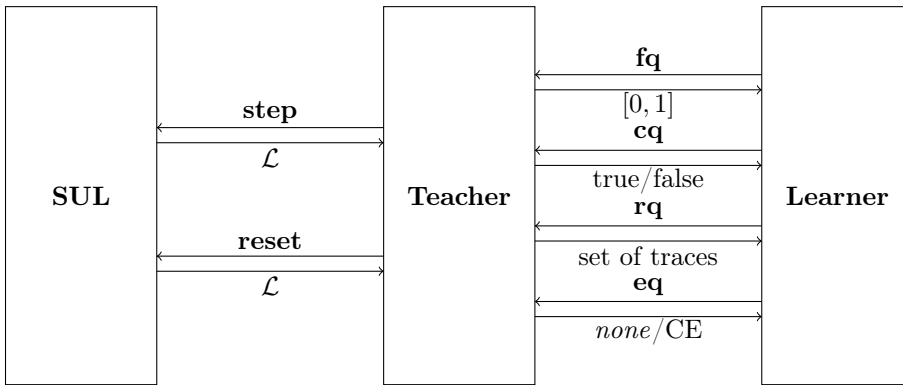


Figure 3.6 – Interactions between the SUL, the teacher and the learner.

In the remainder of this section, we briefly overview the L_{MDP}^* algorithm introduced by Tappler et al in [59].

Let $\mathcal{M} = \langle S, \mathcal{L}, A, \{\tau_a\}_{a \in A}, \pi \rangle$ be the MDP representing the SUL. The L_{MDP}^* algorithm interacts with the SUL through two operations:

- **reset**: resets the SUL to (one of) its initial state: state s is chosen to be the current state with probability $\pi(s)$. Returns $\ell(s)$.
- **step**: executes an action $a \in A$: the current state s is updated s' with probability $\tau_a(s)(s')$. Returns $\ell(s')$.

Throughout the learning process, the teacher will employ these two operations to generate a collection of traces, denoted as \mathbf{T} . With the use of only these two operations and \mathbf{T} , the teacher can respond to the learner's queries.

In L_{MDP}^* , the teacher answers four types of queries:

- frequency query (**fq**): given a trace $o \in \mathbf{Traces}_{\text{fin}}$ and a action a returns the frequency (in \mathbf{T}) of getting ℓ by executing a after observing o , for any label ℓ . Therefore, **fq**($o \cdot a$) is a function $\mathbf{fq}(o \cdot a) : \mathcal{L} \mapsto [0, 1]$
- complete query (**cq**): given a finite trace o and an action a , returns true if sufficient information is available in \mathbf{T} to estimate an output distribution $\mathbf{fq}(o \cdot a)$; returns false otherwise. Hence, **cq** : $\mathbf{Traces}_{\text{fin}} \cdot A \mapsto \mathbb{B}$.
- refine query (**rq**): instructs the teacher to improve its understanding of the SUL by specifically testing it with rarely encountered samples. Traces sampled through **rq** process are incorporated into the existing knowledge set \mathbf{T} , thereby enhancing the accuracy of future probability estimations.
- equivalence query (**eq**): given a hypothesis \mathcal{M} , the **eq** function conducts tests to determine the output-distribution equivalence between the SUL and \mathcal{M} . It returns a counterexample from $\mathbf{Traces}_{\text{fin}}^{\mathcal{M}}$ that demonstrates non-equivalence, or it returns *none* if no counterexample is found. Similar to the approach in active automata learning, we approximate equivalence queries through testing [60]. As a result, if no counterexample is detected, the approximate equivalence queries return *none* instead of *yes*.

Using these four queries, the learner is able to approximate the MDP underlying the SUL. L_{MDP}^* keeps a record of every potential state, denoted by finite traces that lead to them. The **fqs** serve the purpose of approximating transition probabilities. The **cqs** play a role in assessing if there's enough information for a given state. If the conditions are met for a pair of states, it scrutinises whether these two states are statistically identical based on estimated transition probabilities and subsequently merges them. Once all hypothesis states are constructed (according to **cqs**) and no further merges are possible, an **eq** is executed. If no counterexample emerges, the algorithm concludes; otherwise, it employs the counterexample and additional refinement queries **rq**s to identify potential new states, thereby restarting the entire process. A detailed description of the algorithm is given in [59].

As State-merging methods, L_{MDP}^* output models are necessary deterministic, and the user cannot decide a priori the size of the output model. Like Alergia, L_{MDP}^* converges to the SUL when the number of samples the teacher bases its answers from (i.e., \mathbf{T}) converges to infinity.

Chapter 4

Tools

In the previous Chapter, we showed how we can theoretically learn different Markov models. In this Chapter, we present various software packages that enable us to learn these models in practice.

This Chapter is divided in two Sections: the first one is dedicated to AALPY, a Python library implementing Alergia, IOAlergia and L_{MDP}^* , and the second one is dedicated to several other softwares that are worth mentioning.

4.1 AALPY

AALPY, for Active Automata Learning, is a Python library that supports a variety of formalisms, as deterministic finite automata, Mealy machine, MDPs and MCs to name a few [61]. AALPY is developed and maintained by the DES-Lab at Graz University of Technology (Austria). Version 1.0 was released in April 2021, and since then AALPY is updated roughly five times a year. The tool paper describing AALPY got accepted in 2022 [61]. Here, we give an overview of AALPY. A deeper analysis of its performance can be found in Section 7.5.4.

Features AALPY offers 9 learning algorithms for 8 different formalisms, summarised in table 4.1. Most of them are non-stochastic models, which is out of our scope. Our focus will be on stochastic models and their learning algorithms.

```
from aalpy.learning_algs import run_Alergia
type(training_set) # list
output = run_Alergia(training_set, eps=0.1, automaton_type='mc')
type(output) # aalpy.MarkovChain
```

Figure 4.1 – Simple execution of AALPY to learn an MC using Alergia.

Model formalisms	Learning algorithms
DFAs [7]	L^* [33]
Mealy Machines [62]	KV [63]
Moore Machine [62]	RPNI [64]
ONFSMs [65]	L^*_{ONFSM} [65]
Abstracted ONFSMs [66]	L^*_{ONFSM} [66]
MCs	Alergia [46, 47]
MDPs	L^*_{MDP} [67], IOAlergia [49, 68]
SMM [67]	L^*_{SMM} [67]

Table 4.1 – Formalisms and algorithms supported by AALPY

Applications. Despite it has been recently released, AALPY has been used in several contexts [61, 69].

Notably, in [69], the authors propose a general framework to learn a behavioural model of the Bluetooth Low Energy (BLE) protocol implemented by a physical device. Using this framework, they learn a behavioural model of the BLE protocol implemented in five physical devices. Differences in the five implementations were found, thereby indicating the viability of employing automata learning for the purpose of black-box device fingerprinting.

Strengths. The first and main strength of AALPY is its comprehensiveness: AALPY supports a wide variety of models which makes it useful in a wide variety of environments.

Additionally, AALPY has a modular design and can therefore be easily extended to other model formalisms or learning algorithms.

Lastly, the accessibility and usability of AALPY are significantly enriched through its comprehensive documentation and numerous illustrative examples, each showcasing the various functionalities offered by the library. These features facilitate a more user-friendly experience, enabling users to effectively harness the capabilities of AALPY for their specific needs.

Weaknesses. As shown in the Introduction, the application of Markov models is predominantly driven by the necessity to model check the employed models. However, it is worth noting that the sole existing compatibility between AALPY and a model checker resides in the `MDP_2_PRISM_FORMAT` function. Therefore, AALPY is not interoperable with `STORMPY`, it cannot translate other models to `PRISM`, and it does not support models in the `PRISM` format.

4.2 Others

GHMM. The General Hidden Markov Model library (GHMM) is a C library, with a Python interface, that implements data structures and algorithms for HMMs analysis [70]. GHMM supports HMMs and numerous of its extensions, such as GoHMMs. Sadly, GHMM is not maintain anymore. One consequence of this

is that GHMM is only compatible with Python 2.7, whose end-of-life was made official in December 2020, making it hardly usable today. However, the base code is still available (in C). Additionally, ‘GHMM is utterly lacking in documentation’, as we can read on its website.

hmmlearn. HMMLEARN is a widely utilised Python library that implements HMMs along with various extensions and the associated BW algorithms. In particular, HMMLEARN supports HMM and GoHMM. The package includes a documentation with few examples. However, it should be noted that HMMLEARN has not undergone peer review. A brief performance comparison between HMMLEARN and JAJAPY on learning of HMMs is provided in Section 7.5.3.

4.3 Conclusion

Apart from AALPY, there exists no tool that (i) performs the desired task, i.e. it implements at least one of the algorithms described previously, (ii) is peer-reviewed, and (iii) is still easily and widely usable today. For these reasons, we consider in this thesis AALPY as the current state-of-the-art tool for learning stochastic models.

Part II

Contributions

Foreword

This thesis improves the state of the art in stochastic model learning in three ways.

An active version of the BW algorithm for MDPs. This version actively learns MDPs by sampling new traces which are the most informative w.r.t. the current hypothesis. This approach significantly reduces the number of traces required to obtain accurate models, given that we can interact with the SUL. The paper ‘Active Learning of Markov Decision Processes using Baum-Welch algorithm’, presented at the ICMLA’21 [71], introduced this version of the BW algorithm.

A learning algorithm for synchronous compositions of CTMCs. It is very common to represent a system as a synchronous composition of CTMCs. This type of representation is more intuitive and compact. In practice, the Prism language [12] is commonly used for this purpose. However, it is difficult to learn the model resulting from such a composition: firstly, because of the explosion in the number of states, and secondly, because the transitions rates are algebraic compositions of the rates of the compound models.

We propose an algorithm for learning synchronous compositions of CTMCs. Our algorithm is based on an optimisation principle known as the MM algorithm. Since the BW algorithm is an application of the EM algorithm itself, being only an instance of MM, we cannot say that our algorithm is a variant of the BW algorithm. The paper ‘An MM algorithm to Estimate Parameters in Continuous-time Markov Chains’, presented at the QEST’23 [72], introduced this algorithm.

A novel library for stochastic model learning. We have developed JAJAPY, a Python library that implements a wide range of machine learning algorithms for different families of Markov models and being compatible with the model checkers STORM and PRISM. This library makes it easier to experiment with different model architectures and methodologies. Markov models having applications in a wide range of fields beyond machine learning, including finance, biology, speech recognition, natural language processing, and more, JAJAPY could facilitate research in these diverse domains by providing tools to learn Markov models. Finally, its compatibility with model checkers is a valuable feature, especially for safety-critical applications where correctness and reliability are paramount.

The three following Chapters present these three contributions brought by this thesis. Each of them is an adapted version of the original paper introducing these contributions ([71] for the first Chapter, [72] for the second one and [73] for the latter). Notations and vocabulary have been adapted to standardise the final document. Previously defined concepts have been shortened to avoid redundancy.

Chapter 5

Active Learning of Markov Decision Processes using Baum-Welch algorithm

This Chapter revisits and adapts the classic BW algorithm for learning MDPs. We present a model-based active learning sampling strategy that chooses examples which are most informative w.r.t. the current model hypothesis. We empirically compare our approach with state-of-the-art tools and demonstrate that the proposed active learning procedure can significantly reduce the number of observations required to obtain accurate models.

5.1 Introduction

Learning MDPs typically requires more traces as the number of model parameters grows (i.e., transition probabilities to estimate) with the number of actions. While the number of parameters is equal to s^2 for an MC (with s the number of states), this value is equal to $s^2 \cdot a$ for an MDP (with a the number of actions). To address this issue, we employ *active learning*. Rather than collecting data samples at random, we steer the sampling of new traces aiming at uncovering unobserved behaviours, thus improving the accuracy of the current model hypothesis. In this line, we propose to learn an initial hypothesis from a relatively small set of traces sampled at random. Then, for each state in the hypothesis, we compute the expected number of times each action has been chosen from that state. This information is used to devise an observation-based scheduler aimed at restoring balance in the count of actions performed from each hidden state. This helps the collected data set to represent a wider spectrum of the behaviours of the SUL.

Experiments show that our active learning procedure can significantly reduce the number of traces required to obtain accurate models, achieving a faster convergence rate than that observed when employing uniform schedulers.

Other Related Work An influential active automata learning technique is Angluin’s L^* algorithm [33] for learning regular languages, which inspired a number of extensions better suited for modelling reactive systems [57, 60, 74]. In this line of research, Tappier et al. [59] proposed an L^* -based technique for learning (deterministic) MDPs. The method iteratively refines the current hypothesis until the teacher cannot provide a counterexample sequence. For each refinement step a predefined amount of new traces is collected. In contrast to our proposal, new traces are sampled targeting a subset of states that are marked as rare.

Other related work include *model-based* learning techniques for partially observable MDPs (e.g., [75]). These techniques aim at learning how to act in an unknown partially observable domain taking actions based on an approximate model of the domain. Typically, they learn only a portion of the real model that is sufficient to optimise the strategy, leaving unnecessary parts of the system unexplored. In contrast, we aim at learning the whole model and be able to analyse it. This expanded knowledge comes at the cost of increased learning complexity and data requirements.

Synopsis Section 5.2 summarises preliminary concepts and notation; Section 5.3 describes the proposed active learning strategies. Then, Section 5.4 contains an empirical evaluation of our the active learning strategies presented in this Chapter, and Section 5.5 discusses the results and outlines future research directions.

5.2 Preliminaries

In this Chapter, we define *observation-based scheduler*, the family of schedulers that choose the next action according to the trace observed so far, instead of the path. Formally:

Definition 5.2.1 (observation-based scheduler). *A scheduler σ is observation-based if for all $\rho, \rho' \in \mathbf{Paths}_{fin}$ such that $|\rho| = |\rho'|$, $O_{|\rho|} = O_{|\rho'|}$ implies $\sigma(\rho) = \sigma(\rho')$.*

Therefore, we usually define, as a misnomer, observation-based schedulers are as function $\sigma : \mathbf{Traces}_{fin} \mapsto \mathcal{D}(S)$, while, since they are scheduler, they should be define as functions from \mathbf{Paths}_{fin} to $\mathcal{D}(S)$.

We also define *memoryless scheduler*, following the classic definition [7], as a scheduler that chooses the next action according to the last state only. Thus, memoryless schedulers are not observation-based.

Definition 5.2.2 (observation-based scheduler). *A scheduler σ is memoryless-based if for all $\rho, \rho' \in \mathbf{Paths}_{fin}$, $X_{|\rho|} = X_{|\rho'|}$ implies $\sigma(\rho) = \sigma(\rho')$.*

Memoryless schedulers are often defined as function $\sigma : S \mapsto \mathcal{D}(S)$ as a misnomer.

5.3 The Active-BW algorithm

The BW algorithm is a passive learning method: it assumes no interaction with the system, which has to be learned from a fixed set of observations. In situations where one can actively query the system to collect training data, one can think of employing querying strategies to produce new examples that are most informative w.r.t. the systems behaviour. In this way, one can learn qualitatively better models compared to the passive learning approach while collecting a considerably smaller amount of traces.

Let $\mathcal{H} = \langle S, \mathcal{L}, \ell, A, \{\tau_a\}_{a \in A}, \pi \rangle$ and \mathcal{O} be respectively the current hypothesis and the current training set. The active learning procedure iteratively updates \mathcal{H} and \mathcal{O} by performing the following steps:

1. devise an observation-based scheduler from \mathcal{O} and \mathcal{H} ;
2. sample new observation sequences using the above mentioned scheduler, adding them to \mathcal{O} ; and
3. update \mathcal{H} based on the new data using BW.

These steps are repeated until a given sampling budget has been exceeded or no further scrutiny of the system is deemed necessary. Hereafter, we detail how each step is implemented.

We start by computing the matrix $M = (m_{sa})_{s \in S, a \in A}$ where m_{sa} is the expected number of times the action a has been chosen from s , that is computed as follows

$$m_{sa} = \sum_{o \in \mathcal{O}} \sum_{t=1}^{|o|} \mathbb{1}[a_t = a] \gamma_o(s, t), \quad (5.1)$$

then, we define the memoryless scheduler $\sigma_M: S \rightarrow \mathcal{D}(A)$ as

$$\sigma_M(s)(a) = \frac{\sum_{a' \in A} m_{sa'} - m_{sa}}{\sum_{a'' \in A} \sum_{a' \in A} m_{sa'} - m_{sa''}} \quad (5.2)$$

$$= \frac{\sum_{a' \in A} m_{sa'} - m_{sa}}{(|A| - 1) \cdot \sum_{a' \in A} m_{sa'}} \quad (5.3)$$

Intuitively, given the system is in state $s \in S$, the above scheduler chooses an action $a \in A$ with a probability that is opposite to that observed in \mathcal{O} . Since the current state of the system is hidden, when sampling we use a belief state instead. This corresponds to employ the observation-based scheduler σ_M^* defined as follows. For an observation $o = \ell_0 a_0 \cdots \ell_{T-1} a_{T-1} \ell_T \in \mathbf{Traces}_{\text{fin}}$ and an action $a \in A$,

$$\begin{aligned} \sigma_M^*(o)(a) &= \sum_{s \in S} Pr^{\mathcal{H}}[X_T = s | O_T = o] \cdot \sigma_M(s)(a) \\ &= \sum_{s \in S} \gamma_o(s, T) \sigma_M(s)(a). \end{aligned} \quad (5.4)$$

with γ defined in (3.3). Intuitively, the above scheduler works as follows. Having observed o , we believe system is in state $s \in S$ with probability $Pr^{\mathcal{H}}[X_T = s | O_T = o]$; consequently, σ_M^* chooses the action $a \in A$ with probability $\sigma_M(s)(a)$.

The algorithm in Fig. 5.1 describes how we actively sample an observation sequence of length $T \in \mathbb{N}$ emitted by a partially observable MDP \mathcal{M} by using the

```

ACTIVE SAMPLING( $\mathcal{M}, \mathcal{H} = \langle S, \iota, \{\tau_a\}_{a \in A} \rangle, \mathcal{O}, T \in \mathbb{N}$ )
1  Initialise  $M = (m_{sa})_{s \in S, a \in A}$  as Eq. (5.1)
2   $\ell_1 = \text{INIT}(\mathcal{M})$  // initialise the system
3  for each  $s \in S$ 
4       $\alpha(s, 0) = \pi(s)$ 
5  for  $t = 1$  to  $T - 1$ 
6      Sample  $a_t \in A$  according to  $\sum_{s \in S} \frac{\alpha(s, t)}{\sum_{s' \in S} \alpha(s', t)} \sigma_M(s)$ 
7       $\ell_{t+1} = \text{OBSERVE-LABEL}(\mathcal{M}, a_t)$ 
8      for each  $s \in S$ 
9           $m_{sa_t} = m_{sa_t} + \alpha(s, t) / \sum_{s' \in S} \alpha(s', t)$ 
10          $\alpha(s, t + 1) = \sum_{s' \in S} \tau_{a_t}(s')(\ell_{t+1}, s) \cdot \alpha(s', t)$ 
11 // Return the entire observation sequence
12 return  $(\ell_1, a_1) \cdots (\ell_{T-1}, a_{T-1}) \ell_T$ 

```

Figure 5.1 – Active Sampling Strategy

scheduler σ_M^* of Eq. (5.4).

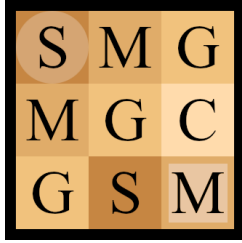
ACTIVE SAMPLING keeps track and updates at each step the matrix M and the current forward distribution $\alpha(\cdot, t) \in \mathcal{D}(S)$. These are respectively used to compute the current belief state $\gamma(\cdot, t) \in \mathcal{D}(S)$ (cf. Eq. (3.3)) and the memoryless scheduler σ_M (cf. Eq. (5.3)), which are used in line 6. After observing an initial label ℓ_0 from the system \mathcal{M} , the initial forward distribution $\alpha(\cdot, 0)$ is computed (lines 3–4). Then, for each time-step t from 0 to $T - 1$, an action $a_t \in A$ is sampled according to σ_M^* , and used to observe the next label ℓ_{t+1} emitted by \mathcal{M} (line 7). The forward distribution $\alpha(\cdot, t + 1)$ and the matrix M are then updated (line 8–10) before moving to the next time-step. The update of the forward probabilities follows Eq. (3.5), while the update of the column vector M_{a_t} follows Eq. (5.1).

5.4 Experimental results

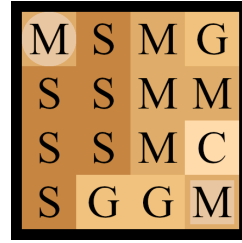
In this section we present an empirical analysis of the active sampling strategy. We will first compare the Active-BW algorithm to the classic BW one, then we compare it to L_{MDP}^* .

5.4.1 Active-BW vs BW

The models. We will use as SUL two variants of the grid world model introduced in [59] (cf. Figure 5.2). A robot is moving in this grid, starting from the top left. The actions are the four directions (north, east, south, and west) and the observed labels represent different terrains. Depending on target terrain the robot may slip and change direction, e.g. move south west instead of south: the probability that the robot slips is respectively 0.0, 0.2, 0.25 and 0.4 if the target terrain is concrete



(a) A 3x3 deterministic grid world



(b) A 4x4 non-deterministic grid world

Figure 5.2 – Grid world models.

(C), grass (G), sand (S) and mud (M). When the robot reaches the goal cell (the bottom right one), it stays there forever, observing *GOAL*. By construction, the 3x3 grid world (*cf.* Figure 5.2a) is deterministic while the 4x4 grid is non-deterministic (*cf.* Figure 5.2b).

The experiment protocol. We compare the active procedure against the passive one and show how the learning accuracy of the former compares to the latter with the size of the training set. The experiments have been performed as follows. Starting from the same initial hypothesis (learned with BW from a small data set) we incrementally grew the data set bigger respectively using the active sampling strategy and a sampling strategy based on a memoryless uniformly distributed selection of actions.

For both models the initial hypothesis was learned from a data set of 50 traces of length 20; then we performed 20 active learning iterations by sampling new traces of length 20. In the following, we consistently employed a trace length of 20. This choice stems from utilising a memoryless uniformly distributed selection of actions, ensuring that the probability of reaching each state of the model within 20 time steps is at least 0.5. In general, this is a reasonable rule of thumb for determining the length of traces used in training and testing the model.

Results. Fig. 5.3 shows the graph of the mean loglikelihood distance with a test set containing 10,000 traces of length 20 paired with standard error bars measured from a number of re-run of the experiment relative to test set for the two models. In our experience, using 10,000 traces of this length is sufficient to achieve accurate loglikelihood distances while maintaining low computation times.

Overall, the graphs in Fig. 5.3 show that the active learning approach provides better approximations than the passive approach. Another interpretation is that the proposed active learning is able to obtain the same level of accuracy than the passive learning approach with a smaller data set. Notably, the graphs show also that the standard error for the active learning method is smaller than the one measured for the passive learning approach. This indicates that our active learning approach is more stable than the passive approach.

As anticipated, the loglikelihood distance in the deterministic 3x3 grid world is

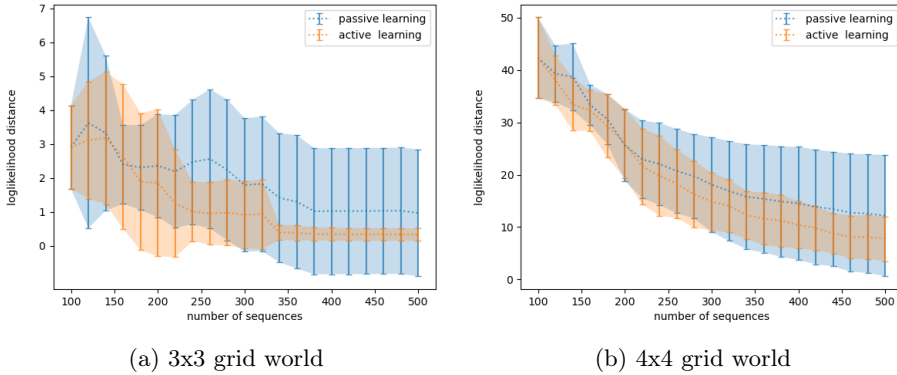


Figure 5.3 – Comparison between the passive learning and active learning procedures: loglikelihood distance relative to a test set of 10,000 traces of length 20.

smaller than in the larger and nondeterministic world. In both cases, we were learning from the same amount of data, but the larger world necessitates the estimation of more transition probabilities than the smaller one.

5.4.2 Active BW vs L_{MDP}^*

We conclude the experiment section by comparing our active learning method against the L_{MDP}^* algorithm [59] for learning deterministic MDPs. We recall that L_{MDP}^* actively refines its current hypothesis as long as the teacher can provide new counterexamples. The implementation of the teacher in the L_{MDP}^* algorithm is done both by checking the conformance and the structure of the hypothesis w.r.t the data set.

The models. We use the same two models for this experiment (the two grid world depicted in Fig. 5.2). The reason is that the 3x3 grid world model is deterministic, and the other is not. However, we know that the models resulting from an execution of L_{MDP}^* can only be deterministic (see Section 3.3). It will therefore be interesting to compare how both L_{MDP}^* and Active-BW behave when SUL is deterministic and when it is not.

The experiment protocol. For this experiment, we use the L_{MDP}^* implementation of AALpy, a Python learning library for automata [61], and JAJAPY (see Chapter 7) for Active-BW.

We start by running L_{MDP}^* for 200 learning iterations on the 3x3 grid world, and for 100 learning iterations on the 4x4 grid.

Then, we run Active-BW on 199 active iterations (plus the initial classic BW one) for the 3x3 grid, and on 99 active iterations for the 4x4 grid, such that the total number of labels used by the two algorithms is as close as possible. In practice, if N is the total number of labels used by L_{MDP}^* , we use, for Active-BW, an original

	true	L_{MDP}^*	Active-BW
overall # of labels	-	69,872	69,860
# of traces	-	15,287	3,493
# of states	17	17	17
loglikelihood distance	0.0	0.724	0.356
$Pr_{\max}(F^{<4}(\text{goal}))$	0.336	0.347	0.325
$Pr_{\max}(\neg G \ U^{<4}(\text{goal}))$	0.072	0.077	0.066

Table 5.1 – L_{MDP}^* vs Active-BW on the 3x3 grid world model.

	true	L_{MDP}^*	Active-BW
overall # of labels	-	818,414	818,400
# of traces	-	132,074	40,920
# of states	27	44	27
loglikelihood distance	0.0	0.144	0.731
$Pr_{\max}(F^{<7}(\text{goal}))$	0.351	0.340	0.360
$Pr_{\max}(F^{<12}(\text{goal}))$	1.0	0.996	1.0
$Pr_{\max}(\neg(C \vee W) \ U^{<7}(\text{goal}))$	0.326	0.312	0.326

Table 5.2 – L_{MDP}^* vs Active-BW on the 4x4 grid world model.

training set containing $\frac{1}{3} \frac{N}{20}$ traces of length 20, and for each of the 199 active iterations, we sample using the active learning strategy $\frac{2}{3} \frac{N}{20 \times 199}$ traces of length 20. Finally, the loglikelihood distance is computed for a test set containing 10,000 traces of length 20.

Results. On the deterministic 3x3 grid, similarities arise between the L_{MDP}^* and Active-BW output models (refer to Table 5.1). Notably, while the Active-BW model demonstrates a smaller loglikelihood distance, the L_{MDP}^* model garners slightly better outcomes in terms of the evaluated properties.

However, the two algorithms manifest disparate performances when shifting focus to the non-deterministic 4x4 grid (see Table 5.2). Specifically, L_{MDP}^* learned a deterministic approximation of the SUL, resulting in a model of nearly twice the magnitude. On one hand, the Active-BW output model closely aligns with the true model across all the checked properties in this experiment. On the other hand, the L_{MDP}^* output model is closer to the SUL in terms of loglikelihood distance.

This divergence could be attributed to the fact that Active-BW is not sensitive to structural counterexamples as the L_{MDP}^* algorithm is. Indeed, when the algorithm encounters a new observation which has probability zero of being generated by the current hypothesis, also the next hypothesis won't be able to generate it. This aspect in particular needs particular attention when learning deterministic models or in general when some observation traces can be emitted only by a single path in the hypothesis model.

5.5 Conclusions and Future Work

In this Chapter we revisited the classic Baum-Welch algorithm for learning models parameters of *nondeterministic* MDPs and Markov chains from a set of traces. Compared with state-of-the-art (passive) learning algorithms like Alergia and IOAlergia, the BW procedure has a higher run-time complexity. However, experiments show that BW is able to learn models that reflect more accurately the behaviours of the observed system. This aspect is more pronounced when learning MDPs from a relatively small set of observations.

Learning model parameters for MDPs typically requires large data sets, especially when the system under learning exhibits a high degree of nondeterminism. To cope with this issue, we proposed a model-based active learning sampling strategy which has three main advantages: (a) it is simple to implement and can be seamlessly integrated into small low power embedded systems; (b) it does not introduce additional overhead with respect to the model update procedure; (c) it collects a diverse and well-spread variety of observations, that better represent the nondeterministic behaviours of the system under learning. Experimental results show that the active procedure strategy outperforms the corresponding passive learning variant in terms of accuracy relative to the size of the data set. This makes our active learning procedure an effective solution when one has the possibility to have limited amount of interactions with the system under learning.

A weakness of our active learning procedure is the fact that is it not sensitive to structural counterexamples. As future work we intend address this issue.

Another interesting research direction consists in generalising the active learning procedure for learning model parameters of stochastic two-player games, allowing one to learn systems that operate in an unknown (adversarial) environment by actively interacting with both players.

Chapter 6

MM Algorithms to Estimate Parameters in Continuous-time Markov Chains

In this Chapter, we address the problem of estimating parameter values of CTMCs expressed as PRISM models from a number of partially-observable executions which might possibly miss some dwell time measurements. The semantics of the model is expressed as a parametric CTMC (pCTMC), i.e., CTMC where transition rates are polynomial functions over a set of parameters. Then, building on a theory of algorithms known by the initials MM, for minorisation–maximisation, we present an iterative maximum likelihood estimation algorithm for pCTMCs.

We present an experimental evaluation of the proposed technique on a number of CTMCs from the quantitative verification benchmark set. We conclude by illustrating the use of our technique in a case study: the analysis of the spread of COVID-19 in presence of lockdown countermeasures.

6.1 Introduction

A continuous-time Markov chain (CTMC) is a model of a dynamical system that, upon entering some state, remains in that state for a random real-valued amount of time —called the dwell time or sojourn time— and then transitions probabilistically to another state. CTMCs are popular models in performance and dependability analysis. They have wide application and constitute the underlying semantics for real-time probabilistic systems such as queuing networks [76], stochastic process algebras [77], and calculi for systems biology [78, 79].

Model checking tools such as PRISM [12] and STORM [13] provide a number of powerful analysis techniques for CTMCs. Both tools accept models written in the PRISM language, a state-based language based on [80] that supports compositionnal design via a uniform treatment of synchronous and asynchronous components.

<pre> ctmc // SIR model parameters const double beta; const double gamma; const double plock; const int SIZE = 100000; // population size module SIR s : [0..SIZE] init 99936; i : [0..SIZE] init 48; r : [0..SIZE] init 16; [] i>0 & i<SIZE & s>0 → beta * s * i * plock / SIZE : (s'=s-1)&(i'=i+1); [] i>0 & r<SIZE → gamma * i * plock : (i'=i-1)&(r'=r+1); endmodule </pre>	<pre> ctmc // SIR model parameters const double beta; const double gamma; const double plock; const int SIZE = 100000; // population size module Susceptible s : [0..SIZE] init 99936; [infection] s>0 → s : (s'=s-1); endmodule module Infected i : [0..SIZE] init 48; [infection] i>0 & i<SIZE → i : (i'=i+1); [recovery] i>0 → i : (i'=i-1); endmodule module Recovered r : [0..SIZE] init 16; [recovery] r<SIZE → r : (r'=r+1); endmodule module Rates [infection] true → beta * plock / SIZE : true; [recovery] true → gamma * plock : true; endmodule </pre>
--	---

Figure 6.1 – (Left) SIR model with lockdown from [81], (Right) Semantically equivalent formulation of the model to the left where different individuals are modelled as distinct modules interacting with each other via synchronisation.

For example, consider the PRISM model depicted in Fig. 6.1 (left) implementing a variant of the Susceptible-Infected-Recovered (SIR) model proposed in [81] to describe the spread of disease in presence of lockdown restrictions. The model distinguishes between three types of individuals: susceptible, infected, and recovered respectively associated with the state variables s , i , and r . Susceptible individuals become infected through contact with another infected person and can recover without outside interference. The SIR model is parametric in β , γ , and $plock$. β is the *infection coefficient*, describing the probability of infection after the contact of a susceptible individual with an infected one; γ is the *recovery coefficient*, describing the rate of recovery of an infected individual (in other words, $1/\gamma$ is the time one individual requires to recover); and $plock \in [0, 1]$ is used to scale down the infection coefficient modelling restrictions to reduce the spread of disease.

Clearly, the outcome of the analysis of the above SIR model is strongly dependent on the parameter values used, as they govern the timing and probability of events of the CTMC describing its semantics. However, in some application domains, parameter values have to be empirically evaluated from a number of partially-observable executions of the model. A paradigmatic example is the modelling pipeline described in [81], where the parameters of the SIR model in Fig. 6.1 (left) are estimated based on a definition of the model as ODEs, and later used in an approximation of the original SIR model designed to reduce the state space of the SIR model in Fig. 6.1 (left). Such modelling pipelines require high technical skills, are error-prone, and are time-consuming, thus limiting the applicability and the user base of model checking tools.

In this work, we address the problem of estimating parameter values of CTMCs expressed as PRISM models from a number of partially-observable executions. The expressive power of the PRISM language brings two technical challenges: (i) the classic state-space explosion problem due to modular specification, and (ii) the fact that the transition rates of the CTMCs result from the algebraic composition of the rates of different (parallel) modules which are themselves defined as arithmetic expressions over the parameters (*cf.* Fig. 6.1). We address the second aspect of the problem by considering a class of *parametric* CTMCs (pCTMCs) [82, 83], which are CTMCs where transition rates are polynomial functions over a fixed set of parameters. In this respect, pCTMCs have the advantage to cover a rich subclass of PRISM models and to be closed under the operation of parallel composition implemented by the PRISM language.

Following the standard approach, we pursue the maximum likelihood estimate (MLE), i.e., we look for the parameter values that achieve the maximum joint likelihood of the observed execution sequences. However, given the non-convex nature of the likelihood surface, computing the global maximum that defines the MLE is computationally intractable [84].

To deal with this issue we employ a theoretical iterative optimisation principle known as MM algorithm [22, 23]. The well-known EM algorithm [21] is an instance of MM optimisation framework and is a versatile tool for constructing optimisation algorithms. MM algorithms are typically easy to design, numerically stable, and in some cases amenable to accelerations [85, 86]. The versatility of the MM principle consists in the fact that is built upon a simple theory of inequalities, allowing one to derive optimisation procedures. The MM principle is useful to derive iterative procedures for maximum likelihood estimation which increase the likelihood at each iteration and converge to some local optimum.

The main technical contribution of this Chapter consists in devising a novel iterative maximum likelihood estimation algorithm for pCTMCs. Crucially, our technique is robust to missing data. In contrast with [48, 87], where state labels and dwell times are assumed to be observable at each step of the observations while only state variables are hidden, our estimation procedure accepts observations to have information to be missing at some steps.

Notably, when state labels and dwell times are observable and only state variables are hidden, our learning procedure results in a generalisation of the Baum-Welch algorithm [26] to pCTMCs.

We demonstrate the effectiveness of our estimation procedure on a case study taken from [81] and show that our technique can be used to simplify modelling pipelines that involve a number of modifications of the model—possibly introducing approximations—and the re-estimation of its parameters.

Related Work Literature on parameter estimation for CTMCs follows two approaches. The first approach is based on Bayesian inference and assumes a probability distribution over parameters which in turn produces an *uncertain* CTMC [88, 89]. In this line of work, Georgoulas et al. [88, 90] proposed ProPPA, a stochastic process algebra with inference capabilities. Using probabilistic inference, the ProPPA model is combined with the observations to derive updated probability

distributions over rates. Uncertain pCTMCS require dedicated model checking techniques [89].

The second approach aims at estimating parameter values producing concrete CTMCS via maximum likelihood estimation. In this line, Geisweiller proposed EMPEPA [87], an expectation-maximisation algorithm that estimates the rate values inside a PEPA model. Wei et al. [91] learn the infinitesimal generator of a continuous-time hidden Markov model by first employing the Baum-Welch algorithm [26] to estimate the transition probability matrix of its (embedded) hidden Markov model from a set of periodic observations.

A large body of literature studies parameter estimation for stochastic reaction networks (SRN) (*cf.* [92, 93] and references therein). According to Gillespie’s theory of stochastic chemical kinetics, SRNs can be represented using CTMCS with states in \mathbb{N}^d . An SRN describes the dynamics of a population of d chemical species by means of a number of *chemical reaction rules*. Notably, the SIR model in Fig. 6.1 was encoded from an SRN. The parameter estimation problem for SRNs focuses on estimating the rate values associated with each reaction rule. In this respect, (i) Andreychenko et al. [94] employs numerical approximations of the likelihood function (and its derivatives) w.r.t. reaction rate constants by dynamically truncating the state space in an on-the-fly fashion, considering only those states that significantly contribute to the likelihood in a given time interval, while (ii) Bayer et al. [95] combines the Monte Carlo version of the expectation-maximisation algorithm [96] with the forward-reverse technique developed in [97] to efficiently simulate SRN bridges conditional on the observed data. Compared with our method, the estimation algorithms of [94, 95] scale better in the number of species and population size, but they assume to observe *all* the coordinates of the state. In our opinion, this limits the applicability of their methods to scenarios where states are partially observed. Such an example is the case study of Section 6.6 where the available data set was only reporting the number of infected individuals (i.e., two components out of three were not observable). Daigle et al. [98] developed an efficient version of the Monte Carlo expectation-maximisation technique which employs modified cross-entropy methods to account for rare events. Notably, their technique is executable also on data sets with missing species but, as for our algorithm, such flexibility comes at the expense of efficiency: the algorithm of [98] took 8.7 days for the parameter estimation on an SRN describing an auto-regulatory gene network with five species, while the method of [95] took 2 days¹.

In contrast with our work, TAlergia (see Section 3.2.2) does not perform parameter estimation over structured models, but learns an unstructured CTMC. Hence, it suits better for learning a single component CTMC when no assumption can be made on the structure or the size of the model.

Another related line of research is parameter synthesis of Markov models [99]. In particular, [100, 83] consider parametric CTMCS, but are generally restricted to a few parameters. In contrast with our work, parameter synthesis revolves around the problem of finding (some or all) parameter instantiations of the model that satisfy a given logical specification.

1. Details on the experiments can be found in the respective papers.

6.2 Preliminaries and Notation

Remark 6.2.1. A CTMC can be equivalently described as a tuple $\langle S, \mathcal{L}, \ell, \rightarrow, \pi \rangle$ where $\rightarrow \subseteq S \times \mathbb{R}_{\geq 0} \times S$ is a transition relation. The transition rate function R induced by \rightarrow is obtained as, $R(s, s') = \sum \{r \mid s \xrightarrow{r} s'\}$ for arbitrary $s, s' \in S$.

The MM Algorithm. The MM algorithm is an iterative optimisation method. The acronym MM has a double interpretation: in minimisation problems, the first M stands for majorise and the second for minimise; dually, in maximisation problems, the first M stands for minorise and the second for maximise. In this paper we only focus on maximising an objective function $f(\mathbf{x})$, hence we tailor the presentation of the general principles of the MM framework to maximisation problems. The MM algorithm is based on the concept of *surrogate function*. A surrogate function $g(\mathbf{x} \mid \mathbf{x}_m)$ is said to *minorise* a function $f(\mathbf{x})$ at \mathbf{x}_m if

$$f(\mathbf{x}_m) = g(\mathbf{x}_m \mid \mathbf{x}_m), \quad (6.1)$$

$$f(\mathbf{x}) \geq g(\mathbf{x} \mid \mathbf{x}_m) \quad \text{for all } \mathbf{x} \neq \mathbf{x}_m. \quad (6.2)$$

In the MM optimisation framework, we maximise the surrogate minorising function $g(\mathbf{x} \mid \mathbf{x}_m)$ rather than the actual function $f(\mathbf{x})$. If \mathbf{x}_{m+1} denotes the maximum of the surrogate $g(\mathbf{x} \mid \mathbf{x}_m)$, then the next iterate \mathbf{x}_{m+1} forces $f(\mathbf{x})$ uphill. Indeed, the inequalities

$$f(\mathbf{x}_m) = g(\mathbf{x}_m \mid \mathbf{x}_m) \leq g(\mathbf{x}_{m+1} \mid \mathbf{x}_m) \leq f(\mathbf{x}_{m+1})$$

follow directly from the definition of \mathbf{x}_{m+1} and the axioms (6.1) and (6.2).

Because piecemeal composition of minorisation works well, the derivations of surrogate functions are typically achieved by applying basic minorisations to strategic parts of the objective function, leaving other parts untouched. Finally, another aspect that can simplify the derivation of MM algorithms comes from the fact that the iterative maximisation procedure hinges on finding $\mathbf{x}_{m+1} = \arg \max_{\mathbf{x}} g(\mathbf{x} \mid \mathbf{x}_m)$. Therefore, $g(\mathbf{x} \mid \mathbf{x}_m)$ can be replaced by any other surrogate function $g'(\mathbf{x} \mid \mathbf{x}_m)$ satisfying $\arg \max_{\mathbf{x}} g(\mathbf{x} \mid \mathbf{x}_m) = \arg \max_{\mathbf{x}} g'(\mathbf{x} \mid \mathbf{x}_m)$ for all \mathbf{x}_m . This is for instance the case when $g(\mathbf{x} \mid \mathbf{x}_m)$ and $g'(\mathbf{x} \mid \mathbf{x}_m)$ are equal up to some (irrelevant) constant c , that is $g(\mathbf{x} \mid \mathbf{x}_m) = g'(\mathbf{x} \mid \mathbf{x}_m) + c$.

6.3 Parametric Continuous-time Markov chains

As mentioned in the introduction, the PRISM language offers constructs for the modular design of CTMCs within a uniform framework that represents synchronous and asynchronous module interaction. For example, consider the PRISM models depicted in Fig. 6.1. The behaviour of each module is described by a set of commands which take the form `[action] guard \rightarrow rate: update` representing a set of transitions of the module. The guard is a predicate over the state variables in the model. The update and the rate describe a transition that the module can make if the guard is true. The command optionally includes an action used to force two or more modules to make transitions simultaneously (i.e., to synchronise). For

example, in the model in Fig. 6.1 (right), in state $(50, 20, 5)$ (i.e., $s = 50$, $i = 20$, and $r = 5$), the model can move to state $(49, 21, 5)$ by synchronising over the action `infection`. The rate of this transition is equal to the product of the individual rates of each module participating in an `infection` transition, which in this case amounts to $0.01 \cdot \text{beta} \cdot \text{plock}$. Commands that do not have an action represent asynchronous transitions that can be taken independently (i.e., asynchronously) from other modules.

By default, all modules are combined following standard parallel composition in the sense of the parallel operator from Communicating Sequential Processes algebra (CPS), that is, modules synchronise over all their common actions. The PRISM language offers also other CPS-based operators to specify the way in which modules are composed in parallel.

Therefore, a parametric representation of a CTMC described by a PRISM model shall consider *transition rate expressions* which are closed under finite sums and finite products: sums deal with commands with overlapping guards and updates, while products take into account synchronisation. In line with [82, 83] we employ *parametric* CTMCs (pCTMCs).

Let $\mathbf{x} = (x_1, \dots, x_n)$ be a vector of parameters. We write \mathcal{E} for the set of polynomial maps $f: \mathbb{R}_{\geq 0}^n \rightarrow \mathbb{R}_{\geq 0}$ of the form $f(\mathbf{x}) = \sum_{i=1}^m b_i \prod_{j=1}^n x_j^{a_{ij}}$, where $b_i \in \mathbb{R}_{\geq 0}$ and $a_{ij} \in \mathbb{N}$ for $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$. \mathcal{E} is a commutative semiring satisfying the above-mentioned requests for transition rate expressions.

Definition 6.3.1. *A pCTMC is a tuple $\mathcal{P} = \langle S, \mathcal{L}, \ell, R, \pi \rangle$ where S , s_0 , and ℓ are defined as for CTMCs, and $R: S \times S \rightarrow \mathcal{E}$ is a parametric transition rate function.*

Intuitively, a pCTMC $\mathcal{P} = \langle S, \mathcal{L}, \ell, R, \pi \rangle$ defines a family of CTMCs arising by plugging in concrete values for the parameters \mathbf{x} . Given a parameter evaluation $\mathbf{v} \in \mathbb{R}_{\geq 0}^n$, we denote by $\mathcal{P}(\mathbf{v})$ the CTMC associated with \mathbf{v} , and $R(\mathbf{v})$ for its rate transition function. Note that by construction $R(\mathbf{v})(s, s') \geq 0$ for all $s, s' \in S$, therefore $\mathcal{P}(\mathbf{v})$ is a proper CTMC.

As for CTMCs, parametric transitions rate functions can be equivalently described by means of a transition relation $\rightarrow \subseteq S \times \mathcal{E} \times S$, where the parametric transition rate from s to s' is $R(s, s')(\mathbf{x}) = \sum \{f(\mathbf{x}) \mid s \xrightarrow{f} s'\}$.

Example 6.3.1. Consider the model in Fig. 6.1 parametric in `beta`, `gamma`, and `plock`. The semantics of this model is a pCTMC with states $S = \{(s, i, r) \mid s, i, r \in \{0, \dots, 10^5\}\}$ and initial state $(99936, 48, 16)$. For example, the initial state has two outgoing transitions: one that goes to $(99935, 49, 16)$ with rate $47.96928 \cdot \text{beta} \cdot \text{plock}$, and the other that goes to $(99935, 48, 17)$ with rate $49 \cdot \text{gamma} \cdot \text{plock}$.

Example 6.3.2. Consider the model in Fig. 6.2 with parameters `mu1a`, `mu1b`, `mu2` and `kappa`. The semantics of this model is a parametric CTMC with states $S = \{(sc, ph, sm) \mid sc, sm \in \{0, \dots, 5\}, ph \in \{1, 2\}\}$ and initial state $(0, 1, 0)$.

The initial state has one outgoing transitions that goes to $(1, 1, 0)$ with rate `lambda` = $4 \cdot c = 20$.

State $(1, 1, 0)$ has three outgoing transitions: one goes to $(2, 1, 0)$ with rate `lambda` = $4 \cdot c = 20$, one to $(1, 2, 0)$ with rate `mu1a`, and one to $(1, 1, 1)$ with rate `mu1b` · 1.

Figure 6.3 is a representation of the Tandem queuing model with `c` = 1 where the synchronous transitions are in red.

```

ctmc
// Tandem Queuing Network [Hermanns, Meyer-Kayser & Siegle]
const int c = 5; // queue capacity
const double lambda = 4 * c;
// model parameters
const double mu1a; const double mu1b; const double mu2; const double kappa;

module serverC
  sc : [0..c] init 0;
  ph : [1..2] init 1;

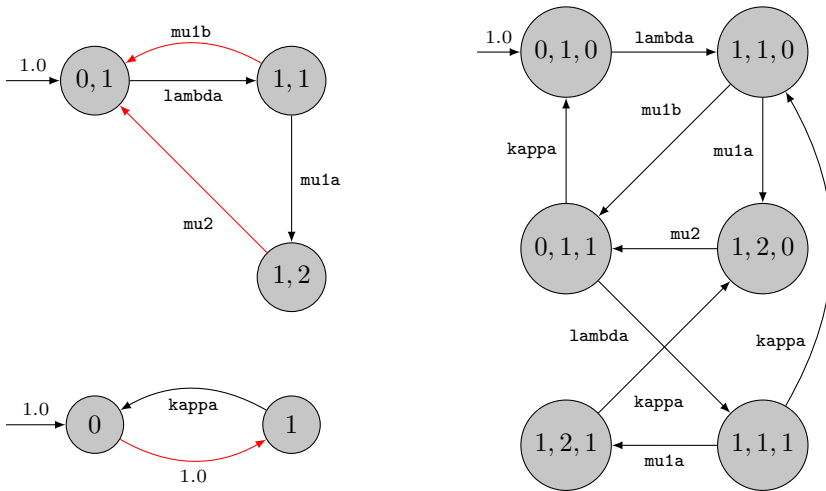
  [] (sc < c) -> lambda : (sc' = sc + 1);
  [route] (sc > 0) & (ph = 1) -> mu1b : (sc' = sc - 1);
  [] (sc > 0) & (ph = 1) -> mu1a : (ph' = 2);
  [route] (sc > 0) & (ph = 2) -> mu2 : (ph' = 1) & (sc' = sc - 1);
endmodule

module serverM
  sm : [0..c] init 0;

  [route] (sm < c) -> 1 : (sm' = sm + 1);
  [] (sm > 0) -> kappa : (sm' = sm - 1);
endmodule

```

Figure 6.2 – Prism model for the tandem queuing network from [101]

Figure 6.3 – (Left) the two components of the Tandem queuing network with $c=1$ (Right) the pCTMC resulting from the synchronous composition of the two models.

One relevant aspect of the class of pCTMCs is the fact that it is closed under parallel composition in the sense described above. This justifies the study of parameter estimation of PRISM models from observed data via maximum likelihood estimation for pCTMCs.

6.4 Estimating Parameters from Partial Observations

In this section, we present an algorithm to estimate the parameters of a pCTMC \mathcal{P} from a collection of i.i.d. timed traces \mathcal{O} . Notably, the algorithm is devised to be robust to missing dwell time values. In this line, we consider *partial observations* of the form $\ell_{0:n}, t_{0:n-1}$ representing a finite sequence $\ell_0 t_0 \cdots t_{n-1} \ell_n$ of consecutive dwell time values and atomic propositions observed during a random execution of \mathcal{M} . Here, as before, the dwell times t_i that are missing are denoted as $t_i = \emptyset$.

We follow a maximum likelihood approach: the parameters \mathbf{x} are estimated to maximise the joint likelihood $l(\mathcal{P}(\mathbf{x}); \mathcal{O}) = \prod_{o \in \mathcal{O}} l(\mathcal{P}(\mathbf{x}); o)$ of the observed data. When \mathcal{P} and \mathcal{O} are clear from the context, we write $l(\mathbf{x})$ for the joint likelihood and $l(\mathbf{x}; o)$ for the likelihood of the observation o .

According to the assumption that some dwell time values may be missing, we recall that the likelihood of a partial observation $o = \ell_{0:n}, t_{0:n-1}$ for a generic CTMC \mathcal{M} is

$$l(\mathcal{M}; o) = \sum_{\substack{\rho \in \mathbf{Paths}(o) \\ \rho = s_0 t_0 \dots s_n}} \left(\pi(s_0) \prod_{i=0}^{n-1} \frac{R(s_i, s_{i+1})}{E(s_i)} \cdot \prod_{i \in \mathcal{T}(\rho)} E(s_i) e^{-E(s_i) t_i} \right). \quad (6.3)$$

Our solution to the maximum likelihood estimation problem builds on the MM optimisation framework [23, 22]. In this line, our algorithm starts with an initial hypothesis \mathbf{x}_0 and iteratively improves the current hypothesis \mathbf{x}_m , in the sense that the likelihood associated with the next hypothesis \mathbf{x}_{m+1} enjoys the inequality $l(\mathbf{x}_m) \leq l(\mathbf{x}_{m+1})$. The procedure terminates when the improvement does not exceed a fixed threshold ϵ , namely when $l(\mathbf{x}_m) - l(\mathbf{x}_{m-1}) \leq \epsilon$.

Before proceeding with the formulation of the surrogate function, we find it convenient to introduce some notation. Let $\mathcal{P} = \langle S, \mathcal{L}, \ell, \rightarrow, \pi \rangle$, we write f_ω for the rate function of a transition $\omega \in \rightarrow$, and write $s \rightarrow \cdot$ for the set of transitions departing from $s \in S$.

Without loss of generality, we assume that the rate function f_ω of a transition is either a constant map, i.e., $f_\omega(\mathbf{x}) = c_\omega$ for some $c_\omega \geq 0$ or a map of the form $f_\omega(\mathbf{x}) = c_\omega \prod_{i=1}^n x_i^{a_{\omega i}}$ for some $c_\omega > 0$ and $a_{\omega i} > 0$ for some $i \in \{1, \dots, n\}$; we write a_ω for $\sum_{i=1}^n a_{\omega i}$. We denote by $\overset{c}{\rightarrow}$ the subset of transitions with constant rate function and $\overset{x}{\rightarrow}$ for the remaining transitions.

To maximise $l(\mathbf{x})$ we propose to employ an MM algorithm based on the following surrogate function $g(\mathbf{x}|\mathbf{x}_m) = \sum_{i=1}^n g(x_i|\mathbf{x}_m)$ where

$$g(x_i|\mathbf{x}_m) = \sum_{\omega \in \overset{x}{\rightarrow}} \xi_\omega a_{\omega i} \ln x_i - \sum_s \sum_{\omega \in s \overset{x}{\rightarrow}} \frac{f_\omega(\mathbf{x}_m) a_{\omega i} \gamma_s}{a_\omega(x_{mi})^{a_\omega}} x_i^{a_\omega}. \quad (6.4)$$

Here the coefficients γ_s and ξ_ω are respectively defined as

$$\gamma_s = \sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i) (\llbracket t_i \neq \emptyset \rrbracket t_i + \llbracket t_i = \emptyset \rrbracket E_m(s)^{-1}) \quad (6.5)$$

$$\xi_\omega = \sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \xi_o(\omega, i) \quad (6.6)$$

where $\gamma_o(s, i)$ denotes the likelihood that having observed o on a random execution of $\mathcal{P}(\mathbf{x}_m)$ the state $X_i = s$, and $\xi_o(\omega, i)^2$ is the likelihood that for such random execution the transition performed from state X_i is ω .

The following theorem states that the surrogate function $g(\mathbf{x}|\mathbf{x}_m)$ is a minoriser of the loglikelihood relative to the observed dataset \mathcal{O} .

Theorem 6.4.1. *The surrogate function $g(\mathbf{x}|\mathbf{x}_m)$ minorises $\ln l(\mathbf{x})$ at \mathbf{x}_m up to an irrelevant constant.*

The proof of this Theorem can be found in Appendix B. By Theorem 6.4.1 and the fact that the logarithm is an increasing function, we obtain that the parameter valuation that achieves the maximum of $g(\mathbf{x}|\mathbf{x}_m)$ improves the current hypothesis \mathbf{x}_m relative to likelihood function $l(\mathbf{x})$.

Corollary 6.4.1. *Let $\mathbf{x}_{m+1} = \arg \max_{\mathbf{x}} g(\mathbf{x}|\mathbf{x}_m)$, then $l(\mathbf{x}_m) \leq l(\mathbf{x}_{m+1})$.*

The surrogate function $g(\mathbf{x}|\mathbf{x}_m)$ is easier to maximise than $l(\mathbf{x})$ because its parameters are separated. Indeed, maximisation of $g(\mathbf{x}|\mathbf{x}_m)$ is done by point-wise maximisation of each univariate function $g(x_i|\mathbf{x}_m)$. This has two main advantages: first, it is easier to handle high-dimensional problems [22, 23]; second, one can choose to fix the value of some parameters and perform the maximisation of $g(\mathbf{x}|\mathbf{x}_m)$ only on the corresponding subexpressions $g(x_i|\mathbf{x}_m)$.

The maxima of $g(x_i|\mathbf{x}_m)$ are found among the *non-negative* roots³ of the polynomial function $P_i: \mathbb{R} \rightarrow \mathbb{R}$

$$P_i(y) = \sum_s \sum_{\omega \in s \xrightarrow{\mathbf{x}}} \frac{f_\omega(\mathbf{x}_m) a_{\omega i} \gamma_s}{(x_{mi})^{a_\omega}} y^{a_\omega} - \sum_{\omega \in \mathbf{x}} \xi_\omega a_{\omega i} \quad (6.7)$$

Remark 6.4.1. *There are some cases when 6.7 admits a closed-form solution. For instance, when the parameter index i satisfies the property $\forall \omega \in \mathbf{x}. a_{\omega i} > 0 \implies a_\omega = C$ for some constant $C \in \mathbb{N}$, then maximisation of $g(x_i|\mathbf{x}_m)$ leads to the following update*

$$x_{(m+1)i} = \left[\frac{(x_{mi})^C \sum_{\omega \in \mathbf{x}} \xi_\omega a_{\omega i}}{\sum_s \sum_{\omega \in s \xrightarrow{\mathbf{x}}} f_\omega(\mathbf{x}_m) a_{\omega i} \gamma_s} \right]^{1/C}$$

A classic situation when the above condition is fulfilled occurs when all transitions ω where x_i appear (i.e., $a_{\omega i} > 0$), the transition rate is $f_\omega(\mathbf{x}) = c_\omega x_i$ (i.e., $a_{\omega i} = a_\omega = 1$). In that case, the above equation simplifies to

$$x_{(m+1)i} = \frac{\sum_{\omega \in \mathbf{x}} \xi_\omega}{\sum_s \sum_{\omega \in s \xrightarrow{\mathbf{x}}} c_\omega \gamma_s}$$

For example, the pCTMC associated with the SIR models in Fig. 6.1 satisfies the former property for all parameters, because all transition rates are expressions

2. $\xi_o(\omega, i) = \xi_o(s, i)(s')$, with ω the transition from s to s'

3. Note that P_i always admits non-negative roots. Indeed, $P_i(0) \leq 0$ and $P_i(M) > 0$ for $M > 0$ sufficiently large. Therefore, by the intermediate value theorem, there exists $y_0 \in [0, M)$ such that $P_i(y_0) = 0$.

either of the form $c \cdot \text{plock} \cdot \text{beta}$ or the form $c \cdot \text{plock} \cdot \text{gamma}$ for some constant $c > 0$. Furthermore, if we fix the value of the parameter `plock` the remaining parameters satisfy the latter property. In Section 6.6, we will take advantage of this fact for our calculations. \square

Finally, we show how to compute $\gamma_o(s, i)$ and $\xi_o(\omega, i)$ w.r.t. the observation $o = \ell_0 t_0 \cdots t_{|o|-1} \ell_{|o|}$ by using standard forward and backward procedures. We define the forward function $\alpha_o(s, i)$ and the backward function $\beta_o(s, i)$ respectively as

$$\begin{aligned} \alpha_o(s, i) &= l(Y_{0:i} = \ell_0 \dots \ell_i, T_{0:i-1} = t_0 \dots t_{i-1}, X_i = s \mid \mathcal{P}(\mathbf{x}_m)), \text{ and} \\ \beta_o(s, i) &= l(Y_{i:T} = \ell_i \dots \ell_T, T_{i:T-1} = t_i \dots t_{T-1} \mid X_i = s, \mathcal{P}(\mathbf{x}_m)). \end{aligned}$$

These can be computed using dynamic programming according to the following recurrences. Let $\mathcal{P}(\mathbf{x}_m) = \langle S, \mathcal{L}, \ell, R, \pi \rangle$, then

$$\alpha_o(s, i) = \begin{cases} \pi(s) \phi_o(s, i) & \text{if } i = 0 \\ \phi_o(s, i) \sum_{s' \in S} \frac{R(s', s)}{E(s')} \alpha_o(s', i-1) & \text{if } 0 < i \leq |o| \end{cases} \quad (6.8)$$

$$\beta_o(s, i) = \begin{cases} 1 & \text{if } i = |o| \\ \sum_{s' \in S} \frac{R(s, s')}{E(s)} \beta_o(s', i+1) \phi_o(s', i+1) & \text{if } 0 \leq i < |o| \end{cases} \quad (6.9)$$

where

$$\phi_o(s, i) = \begin{cases} \llbracket \ell(s) = \ell_i \rrbracket E(s) e^{-E(s)t_i} & \text{if } 0 \leq i < |o| \text{ and } t_i \neq \emptyset \\ \llbracket \ell(s) = \ell_i \rrbracket & \text{if } t = |o| \text{ or } t_i = \emptyset. \end{cases} \quad (6.10)$$

Finally, for $s \in S$ and $\omega = (s \xrightarrow{f_\omega} s')$, $\gamma_o(s, i)$ and $\xi_o(\omega, i)$ are related to the forward and backward functions as follows

$$\gamma_o(s, i) = \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{s' \in S} \alpha_o(s', i) \beta_o(s', i)}, \quad (6.11)$$

$$\xi_o(\omega, i) = \frac{\alpha_o(s, i) f_\omega(\mathbf{x}_m) \phi_o(s', i+1) \beta_o(s', i+1)}{E(s) \sum_{s'' \in S} \alpha_o(s'', i) \beta_o(s'', i)}. \quad (6.12)$$

The case of non-timed observations. Consider the limit situation when dwell time variables are not observable (i.e., for all traces, $t_i = \emptyset$ for all $i = 1 \dots |o| - 1$). Under this assumption, two CTMCs \mathcal{M}_1 and \mathcal{M}_2 having the same embedded Markov chain satisfy $l(\mathcal{M}_1; \mathcal{O}) = l(\mathcal{M}_2; \mathcal{O})$. In other words, when dwell time variables are not observable the MLE objective does not fully capture the continuous-time aspects of the model under estimation.

The next section provides experimental evidence that, when the number of parametric transitions is sufficiently small relative to that of constant transitions, our algorithm can hinge on the value of the transition rates that are fixed, leading the procedure to converge to the real parameter values.

6.5 Experimental evaluation

We implemented the algorithm from Section 6.4 as an extension of the JAJAPY Python library (see Chapter 7) [73], which has the advantage of being compatible with PRISM models. In this section, we present an empirical evaluation of the efficiency of our algorithm as well as the quality of their outcome. To this end, we employ a selection of CTMCs from the QComp benchmark set [1]. Experiments on each model have been designed according to the following setup.

For each model, we selected a set of parameters to be estimated as well as the set of observable atomic propositions⁴. We then estimated the parameter values from a training set consisting of 100 observation sequences of length 30, generated by simulating the original benchmark model. As in the previous chapter, the traces are long enough to explore the state space. We chose to use only 100 traces to highlight the efficiency of our method in terms of input data and to maintain a low computation time. After the estimation, we verify all the formulas associated with the given benchmark model and compare the result with the expected one.

We perform experiments both using timed and non-timed observations. Each experiment is repeated 10 times by randomly re-sampling the initial parameter values \mathbf{x}_0 in the range $[0.00025, 0.0025]$. We annotate the running time, the relative error δ_i for each parameter x_i , and the relative error Φ_i for each formula⁵.

Model	$ \mathcal{M} $	$ \rightarrow $	$ p $	Timed Observations				Non-timed Observations			
				Time(s)	Iter	avg δ	avg Φ	Time(s)	Iter	avg δ	avg Φ
polling	240	800	2	136.430	4	0.053	0.421	33.743	12	1.000	7.146
cluster	276	1120	3	132.278	3	0.089	1.293	279.853	12	0.313	3.827
tandem	780	2583	4	1047.746	3	0.043	0.544	4302.197	74	0.161	1.354
philosophers (i)	1065	4141	3	2404.803	3	0.043	0.119	2232.706	6	0.263	0.235
philosophers (ii)	1065	4141	4	9865.645	12	0.032	0.026	33265.151	200	0.870	2.573

Table 6.1 – Performance comparison on selected QComp benchmarks [1].

Table 6.1 reports the aggregated results of the experiments. The columns $|\mathcal{M}|$, $|\rightarrow|$ and $|p|$ provide respectively the number of states and transitions of the model and the number of parameters to estimate; the columns “Time” and “Iter” respectively report the average running time⁶ and number of iterations; and the columns “avg δ ” and “avg Φ ” respectively report the average relative error of the estimated parameters and model checking outcomes. Unsurprisingly, the quality of the estimation is higher for timed observations. Despite in most cases the initial parameter valuation \mathbf{x}_0 being picked far from the real parameter values, our method is capable to get close to the expected parameter values by using relatively few observation sequences. Most of the formulas employed in the experiments compute expected

4. The models are available at github.com/Rapfff/MM-PCTMC-benchmark-models. The source files contain a description of the parameters and what is observable.

5. The relative error is $|e - r|/|r|$, where e (resp. r) is the estimated (resp. real) value.

6. Experiments were performed on a Linux machine with an AMD-Ryzen 9 3900X 12-Core processor and 32 GB of RAM.

accumulated rewards for a time horizon exceeding that of the used training set, as a consequence, also the error tends to build up. The issue can be tamed by having longer observations in the training set. Notably, for timed observations, each iteration is more expensive than non-timed ones, but the additional overhead is largely compensated by a consistently smaller number of iterations. Interestingly, the number of iterations from timed to non-timed training set seems to grow exponentially with the number of parameters to estimate.

To understand how, for non-timed observation, the quality of the estimation varies based on the number of constant transitions we ran our algorithm on two variants of the `philosophers` model: (i) with the variable `gammax` as a constant; and (ii) with `gammax` as a parameter. The algorithm clearly benefits from the presence of constant transitions and it converges way faster to better estimates.

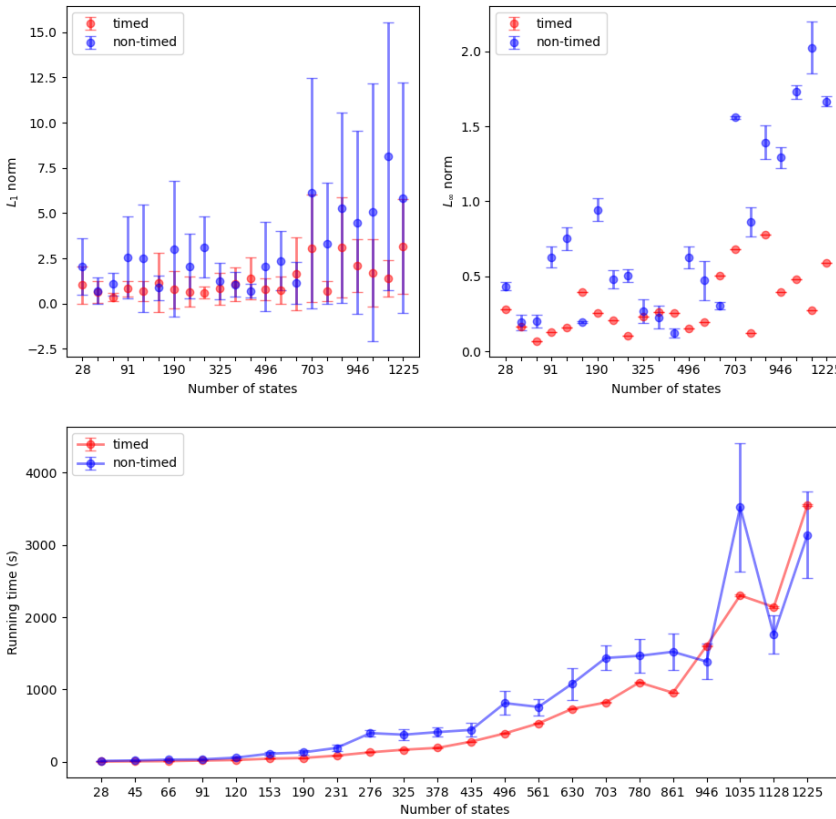


Figure 6.4 – Comparison of the performance of the estimation for timed and non-timed observations on the tandem queueing network with different size of the queue.

Fig. 6.4 reports the results of the experiments performed on the tandem queueing network model from [101] for different sizes of the queue. Each experiment was repeated 10 times by randomly re-sampling the initial valuation \mathbf{x}_0 in the interval $[0.1, 5.0]$. Accordingly, measurements are presented together with their respective

error bars. The graph of the running time (*cf.* Fig. 6.4 bottom) follows a quadratic curve in the number of states both for timed and non-timed observations. However, for non-timed observations, the variance of the measured running times tends to grow with the size of the model. Fig. 6.4 (top) shows how the L_1 -norm (resp. L_∞ -norm) of the vector $\delta = (\delta_i)$ may vary for different size of the model. The variance of the measured relative errors is larger in the experiments performed with non-timed observations. Notably, for timed observations, the quality of the estimation remained stable despite the size of the model increased relative to the size of the training set. This may be explained by the fact that, in the tandem model, the parameters occur in many transitions.

6.6 Case Study: SIR modelling of pandemic

In this section, we take as a case study the modelling pipeline proposed by Milazzo [81] for the analysis and simulation in PRISM of the spread of COVID-19 in presence of lockdown countermeasures. The modelling pipeline includes: (i) parameter estimation from real data based on a modified SIR model described by means of a system of ODEs; (ii) encoding of the modified SIR model into as a PRISM model; and (iii) stochastic simulation and model checking with PRISM.

The model devised in step (ii) is depicted in Fig. 6.1 (left). However, to perform the analysis, Milazzo had to apply “a couple of modelling tricks (variable pruning and upper bounds) that allowed state space of the model [...] to be reduced by several orders of magnitude.” [81]. These kinds of modelling tricks are not uncommon in formal verification, but they require the modeler to ensure that the parameter values estimated for the original model are still valid in the approximated one. In this section, we showcase the use of our algorithm to simplify this task. Specifically, we generate two training sets by simulating the SIR model in Fig. 6.1 using PRISM and, based on that, we re-estimate `beta`, `gamma`, and `plock` on an approximated version of the model (*cf.* Fig. 6.5).

```

ctmc
// bounds
const int ubound_i; const int lbound_i; const int nb_r = 10;
const double size_r = 500/nb_r; const int SIZE = 100000;
const double beta; const double gamma; const double plock; // SIR model parameters

module SIR
  i : [lbound_i..ubound_i] init 48;
  r : [0..nb_r - 1] init 0;

  [infection] i>0 & i<ubound_i → i * (SIZE - (i + (r + 0.5) * size_r)) : (i'=i + 1);
  [recovery] i>0 & r<nb_r - 1 → i * ((size_r) - 1)/(size_r) : (i'=i - 1);
  [recovery] i>0 & r<nb_r - 1 → i * 1/(size_r) : (r'=r + 1) & (i'=i - 1);
  [recovery] i>0 & r=nb_r - 1 → i : (i'=i - 1);
endmodule

module Rates
  [infection] true → beta * plock/SIZE : true;
  [recovery] true → gamma * plock : true;
endmodule

```

Figure 6.5 – Approximated SIR model.

Parameter	Expected Value	Estimated Value	Absolute Error
beta	0.122128	0.135541	0.013413
gamma	0.127283	0.128495	0.001212
plock	0.472081	0.437500	0.034581

Table 6.2 – Parameter estimation on the approximated SIR model.

The first training set represents the spread of the disease without lockdown (i.e., `plock` = 1), while the second one is obtained by fixing the value of `plock` estimated in [81] (i.e., `plock` = 0.472081). In line with the data set used in [81], both training sets consist of one (timed) observation reporting the number of infected individuals for a period of 30 days.

The estimation of the parameters `beta`, `gamma` and `plock` is performed on the model depicted in Fig. 6.5. As in [81], we use an approximated version of the original SIR model (*cf.* Fig. 6.1) obtained by employing a few modelling tricks: variable pruning, set upper bounds on the state variable `i`, and re-scaling of the variable `r` in the interval $[0, \text{nb_r} - 1]$. These modelling tricks have the effect to reduce the state space of the underlying CTMC, speeding-up in this way parameter estimation and the following model analysis.

We perform the estimation in two steps. First, we estimate the values of `beta` and `gamma` on the first training set with `plock` set to 1 (i.e., with no restrictions). Then, we estimate the value of `plock` on the second training set with `beta` and `gamma` set to the values estimated in the first step. Each step was repeated 10 times by randomly re-sampling the initial values of each unknown parameter in the interval $[0, 1]$. Table 6.2 reports the average estimated values and absolute errors relative to each parameter. The running time of each estimation was on average 89.94 seconds⁷. Notably, we were able to achieve accurate estimations of all the parameters from training sets consisting of a single partially-observable execution of the original SIR model. As observed in Section 6.5, this may be due to the fact that each parameter occurs in many transitions.

This case study demonstrates that our estimation procedure can be effectively used to simplify modelling pipelines that involve successive modifications of the model and the re-estimation of its parameter values.

6.7 Conclusion and Future Work

We presented a novel technique to estimate parameter values of CTMCs expressed as PRISM models from partially-observable executions. We demonstrated, with a case study, that our solution is a concrete aid in applications involving modelling and analysis, especially when the model under study requires successive approximations that require re-estimation of the parameters. The major strengths of our algorithm are (i) its interoperability with the model checking tools PRISM and

⁷. Experiments were performed on a Linux machine with an AMD-Ryzen 9 3900X 12-Core processor and 32 GB of RAM.

STORM, and (ii) the fact that it accepts partially-observable data sets where both state and dwell times can be missing. However, the generality of our approach comes at the expense of efficiency. The computations of the forward and backward functions which are required to update the coefficients of the surrogate function 6.4 have a time and space complexity that grows quadratically in the number of states of the pCTMC, thus limiting the number of components that our implementation can currently handle. In future work, we consider investigating how to speed up the computation of the forward and backward functions either by integrating GPU-accelerated techniques from [102] or by replacing their exact computation in favor of numerical approximations obtained through Monte Carlo simulations in line with the idea employed in Monte Carlo EM algorithm [96].

Notably, the algorithm presented in this paper was devised following simple optimisation principles borrowed from the MM optimisation framework. We suggest that similar techniques can be employed to other modelling languages (e.g., Markov automata [103, 104]) and metric-based approximate minimisation [105, 106]. An interesting future direction of research consists in extending our techniques to MDPs by integrating the active learning strategies [71].

Chapter 7

Jajapy: a learning library for stochastic models

We present JAJAPY, a Python library that implements a number of methods to aid the modelling process of Markov models from a set of partially-observable executions of the system. Currently, JAJAPY supports different types of Markov models such as discrete and continuous-time Markov chains, Markov decision processes, hidden Markov models and Gaussian observation hidden Markov models.

JAJAPY can be used both to learn the model from scratch or to estimate parameter values of a given model so that it fits the observed data the best. To this end, the tool offers different learning techniques, either based on expectation-maximization or state-merging methods, each adapted to different types of Markov models. One key feature of JAJAPY consists in its compatibility with the model checkers STORM and PRISM.

This Chapter briefly presents JAJAPY’s functionalities and reports an empirical evaluation of their performance and accuracy. We conclude with an experimental comparison of JAJAPY against AALPY, which is the current state-of-the-art Python library for learning automata. JAJAPY and AALPY complement each other, and the choice of the library should be determined by the specific context in which it will be used.

JAJAPY’s source code follows a modular architecture design and can therefore be extended to other modelling formalisms and learning algorithms. JAJAPY’s documentation can be found on Read the Docs [107] that is complemented with a short video-introduction available on Zenodo [108].

7.1 Introduction

Markov models are a very popular formalism. Discrete-time Markov chains (MCs) and continuous-time Markov chains (CTMCs) have wide applications in performance and dependability analysis, whereas Markov decision processes (MDPs) are key models for stochastic decision-making and planning which find numerous

applications in the design and analysis of cyber-physical systems.

PRISM [12] and STORM [13] are two widely-used model checking tools that provide an efficient and reliable way to verify the correctness of probabilistic systems. They both accept models written in the PRISM language, an expressive state-based language based on [80]. PRISM is a powerful tool for modeling and analysing MCs, MDPs, and probabilistic timed automata. It has a user-friendly interface and supports a variety of analysis techniques, including model checking, parameter synthesis, and probabilistic model checking. STORM, on the other hand, is a highly scalable and efficient tool for analysing probabilistic systems with continuous-time and hybrid dynamics [109]. It supports both explicit and symbolic model representation, and provides state-of-the-art algorithms for model checking and synthesis tasks. Both tools have been extensively used in academia and industry to analyse a wide range of systems, including communication protocols, cyber-physical systems, and biological systems.

The standard assumption of model checking tools is that the model is known precisely. For many application domains, this assumption is too strong. Often the model is not available, or at best is partially known. In such cases, the model is typically estimated empirically from a set of partially-observable executions (a.k.a. traces). Depending on the system under consideration, traces may be collected offline in the form of time series or (possibly continuous) streams of system logs, or the modeller can actively query the system and stir the exploration of its dynamics. In the latter situation, interaction with the system may be limited due to safety critical concerns, or simply to comply with the budget allocated for the task.

To effectively exploit the characteristics of different learning scenarios it is convenient to have a single library that provides a variety of learning algorithms, which can handle different learning scenarios and model types seamlessly, while integrating well with the model-and-verification workflow of PRISM and STORM.

In this Chapter, we present JAJAPY [73, 110], a free open-source Python library that offers a number of techniques to learn Markov models from traces and is interoperable with PRISM and STORM. JAJAPY implements the following machine-learning techniques:

- (i) ALERGIA [46, 47] and IOALERGIA [49, 111], passive learning procedures that learn respectively MCs and (deterministic) MDPs from a set of traces by successively merging compatible states;
- (ii) a number of adaptations of the Baum-Welch algorithm [26] to learn MCs, MDPs [71], HMMs [26], GoHMMs [112] and CTMCs [72] by estimating their transition probabilities given a set of traces and the size of the resulting model;
- (iii) active learning strategies to enhance the quality of the MDPs learned using the Baum-Welch algorithm [71] when the user has the possibility to interact with the system (see Chapter 5);
- (iv) MM algorithms [72] for estimating parameter value in parametric CTMCs (pCTMCs) from a set of (possibly non-timed) traces (see Chapter 6).

JAJAPY implements also metrics to independently evaluate the output model against a test set. This is particularly useful to measure the degree of generalisation that

the output model offers on top of the training set and assess whether the output model overfits the training data or not.

Interoperability with PRISM and STORM is achieved by supporting import and export functions for PRISM models as well as STORMPY sparse models.

Related work. AALPY [61] is a recent Python library that can learn both non-stochastic and stochastic models. In particular, AALPY can learn MDPs using L_{MDP}^* [67], an extension of Angluin’s L^* algorithm [33], and MCs using Alergia [46, 47]. In Section 7.5.4, we compare JAJAPY and AALPY performance.

Other automata learning frameworks have been developed as well. For example, Learnlib [113] and libalf [114], which learn non-stochastic models. In contrast with these tools, as of now, JAJAPY primary focus is on learning Markov models.

In this Chapter, we present the different sampling methods implemented in Jajapy and AALpy. However, other methods also exist, such as the MDI algorithm [51] and [115], two state-merging based approaches, or the Bayesian method using Gibbs sampling [116] proposed by Neal in [34].

MDPs are extensively used in *reinforcement learning* as in [117, 118, 119], and in *robust reinforcement learning* [120] as in [121, 122, 123]. In this context, the objective is to learn an optimal policy that maximises long-term rewards in a given environment.

A related line of research is model synthesis. Counterexample-guided inductive synthesis (CEGIS) [124] study the problem of completing a given program sketch (i.e., a probabilistic program with holes) so that it satisfies a given set of quantitative specifications. Another approach is parameter synthesis [125, 126], where the objective is to find some (or all) instances of a given parametric Markov model satisfying a logic formula. In [127], the authors combine parameter synthesis and parametric inference techniques to synthesize feasible parameter valuations and quantify the confidence that the corresponding model satisfies a given property of interest.

Outline. We start with a quick introduction to JAJAPY functionalities in Section 7.2, then we explain JAJAPY’s features from a theoretical perspective in Section 7.3. In Section 7.4, we present some technical aspects of JAJAPY, and in Section 7.5 we evaluate our tool and compare it to AALPY.

7.2 Jajapy in a nutshell

JAJAPY offers learning methods to construct an accurate model of a system under learning (SUL) from a set of traces and export it to a format that can be directly used in STORM and PRISM for analysis. All models supported by JAJAPY can be imported from and exported to PRISM and STORMPY, except HMMs and GoHMMs: HMMs being equivalent to MCs (see Theorem 2.5.1), they can indirectly been exported to model checkers (after being converted to MCs); and GoHMMs are not compatible with these two model checkers since they do not support model checking for this family of Markov models.

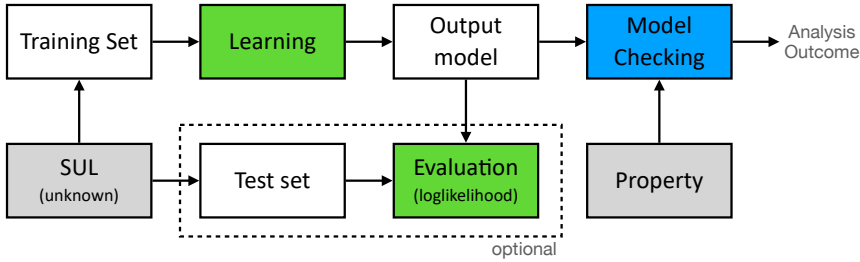


Figure 7.1 – A complete modeling and verification workflow using JAJAPY . The phases where JAJAPY is employed are highlighted in green, whereas the phase in blue is assumed to be performed with STORM or PRISM.

```

from jajapy import BW
type(training_set) # list
output_model = BW().fit(training_set, nb_states=10)
type(output_model) # stormpy.SparseDtmc
  
```

Figure 7.2 – Simple execution of JAJAPY BW to learn an MC with 10 states.

In the following, we call *training set* (resp. *test set*) the collection of traces used to learn the SUL model (resp. to evaluate the learning output model). Depending on the nature of the training set, JAJAPY learns different types of models: (i) MCs and HMMs are learned from sequences of labels (i.e., sequences of atomic propositions), (ii) CTMCs and pCTMCs are learned from times series of labels, (iii) MDPs are learned from alternating sequences of actions and labels, and (iv) GoHMMs are learned from sequences of vectors of n real values, n being fixed for all the sequences. A trace denotes, depending on the context, a sequence of labels, a time series of labels, an alternating sequence of actions and labels, or a sequences of vectors of real numbers. The length of a trace is always the number of labels (or vectors) it contains.

The first and main learning algorithm offered by JAJAPY is the Baum-Welch (BW) algorithm. It takes as input a training set and an initial hypothesis, i.e. a Markov model. During the BW execution, the transition probabilities of this initial hypothesis will be updated but no state will be added/removed from it. Therefore, the number of states in the resulting model will be equal to the number of states in the initial hypothesis. By default JAJAPY generates a random initial hypothesis (given as input the number of states) but the user can also provide one explicitly. This enables the user to exploit his knowledge of the SUL to enhance the learning process. Such an initial hypothesis can be a JAJAPY model, or a STORMPY sparse model or a model saved in a PRISM file if the initial hypothesis is neither an HMM nor a GoHMM. Given a training set and an initial hypothesis, the BW algorithm constructs an approximate representation of the SUL, called *output model*.

As an alternative to the BW algorithm, JAJAPY offers implementations of Alergia and IOAlergia to learn respectively MCs and MDPs. These algorithms take as

input the training set and a *confidence* parameter.

Once JAJAPY has produced the output model, the user can use STORMPY to verify the model against some properties of interest supported by STORM. The output model can also be exported to a PRISM model and analysed with the PRISM model checker.

7.3 Learning probabilistic models

In this Section, we briefly describe the key characteristics of the learning methods for Markov models currently available in JAJAPY and AALPY.

These methods belong to two categories, active and passive. *Active* learning methods learn from interactions with the SUL, while *passive* methods learn from the training set only. Active learning methods are usually more efficient (in terms of data), but can be used only if it is possible to interact with the SUL.

Some learning methods allow the user to decide the size (i.e. the number of states) of the output model, preventing the algorithm from generating models too large to be efficiently analysed. The downside of such a feature consists in the fact that, if the number of states requested is too large (resp. small), the output model may overfit (resp. underfit) the training set.

Some of the learning methods described below assume the Markov model underlying the SUL to be deterministic (see definition 2.4.1). When such methods are exercised with a SUL that is non-deterministic, they are not guaranteed to converge to the true model, instead, they will return a deterministic model that approximates the SUL. Typically, the approximated model is larger than the SUL.

Expectation Maximisation approach. The Baum-Welch (BW) algorithm is an iterative maximum likelihood estimation method to estimate the parameters of Markov models [25]. This technique is an application of the Expectation Maximisation algorithm. Originally designed for Hidden Markov Models [26], it has been adapted to MCs, CTMCs, MDPs and GoHMMs [71, 72, 112].

Given a set of traces \mathcal{O} (the training set) and an initial hypothesis \mathcal{H}_0 , the BW algorithm iteratively updates \mathcal{H}_0 such that the likelihood that the hypothesis generates \mathcal{O} has increased with respect to the previous step. The algorithm stops when the likelihood difference between two successive hypotheses is lower than a fixed threshold ϵ . In JAJAPY, the user can also set an upper bound on the number of BW iterations. BW converges to a local optimum [128].

The BW algorithm is a passive learning approach, it allows the user to decide the size of the output model, and can learn non-deterministic models.

Active learning with sampling strategy. JAJAPY implements an active learning extension of the BW algorithm for MDPs [71]. This method uses a sampling strategy to generate new training samples that are most informative for the current model hypothesis. With this method, the user decides the size of the output model. This algorithm is able to learn non-deterministic models.

Currently, JAJAPY only supports the sampling strategy described in [71].

Algorithm	Model	Reference	Active	# states	Non-det.	JAJAPY	AALPY
BW-MC	MC	[71]	✗	✓	✓	✓	✗
BW-CTMC	CTMC	[72]	✗	✓	✓	✓	✗
BW-HMM	HMM	[26]	✗	✓	✓	✓	✗
BW-GoHMM	GoHMM	[112]	✗	✓	✓	✓	✗
MM-pCTMC	pCTMC	[72]	✗	✓	✓	✓	✗
BW-MDP	MDP	[71]	✗	✓	✓	✓	✗
Active-BW	MDP	[71]	✓	✓	✓	✓	✗
Alergia	MC	[46, 47]	✗	✗	✗	✓	✓
IOAlergia	MDP	[49, 68]	✗	✗	✗	✓	✓
L_{MDP}^*	MDP	[67]	✓	✗	✗	✗	✓

Table 7.1 – Key characteristics of the selected learning algorithms for Markov models.

State-merging approach. Both JAJAPY and AALPY provide an implementation of the Alergia algorithm [46, 47] to learn MCs and its extension IOAlergia [49] to learn MDPs. These algorithms use a state-merging approach. Starting from a maximal tree-shaped probabilistic automaton representing the training set, they iteratively merge states that are ‘similar enough’ according to an Hoeffding test [52]. The accuracy of the Hoeffding test is provided as input. These algorithms are passive, they do not allow the user to choose the number of states in the output model, and they assume the SUL to be deterministic.

Active learning with membership and equivalence queries. AALPY provides an implementation of L_{MDP}^* [67], an extension of Angluin’s L^* algorithm [33] to learn MDPs. As for Alergia, this method assumes the SUL to be deterministic, and the size of the output model cannot be chosen in advance.

Table 7.1 summarises the key characteristics of the learning methods discussed above. The 5th column indicates whether or not the user can choose the number of states in the output model, and the 6th column indicates whether the algorithm is able to generate non-deterministic models or not.

7.4 Architecture and technical aspects

In this Section, we describe some internal aspects of JAJAPY.

Jajapy models. In the following we describe the structure and the relations between the different model classes in Jajapy (see the UML diagram in Figure 7.3).

There exist three families of models in JAJAPY, each of them represented by its abstract class: `Base_MC`, corresponding to the models where the states are labelled (MCs, MDPs, CTMCs), `Base_HMM`, corresponding to the models where the states

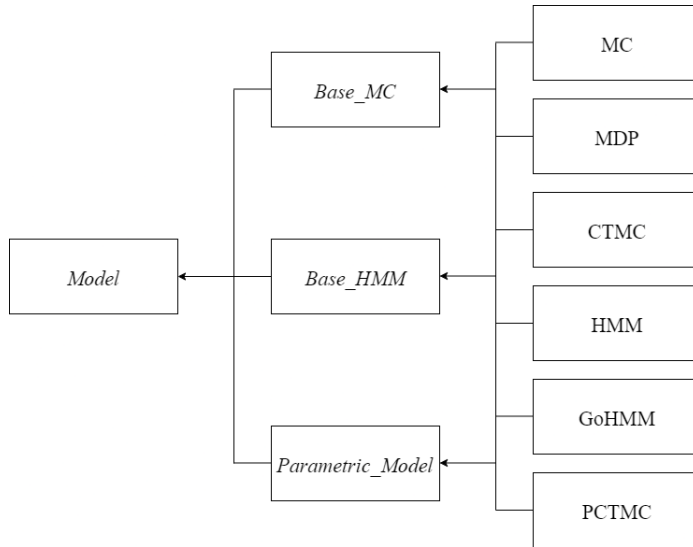


Figure 7.3 – Class diagram for JAJAPY model classes.

are associated to a probability distribution over the labels/observations (HMM, GoHMMs), and `Parametric_Model`, where the transitions probabilities/rates are polynomial functions over a set of parameters (pCTMCs). These abstract classes are inherited by the six classes dedicated to the six kind of Markov models currently supported by JAJAPY. Each of these three abstract classes inherits from an abstract class `Model`, that implements the methods to run the model, generate traces and compute the loglikelihood of a set of traces under the model. This class contains also an attribute `matrix` which contains the transition probabilities/rates. This `matrix` is a *Numpy ndarray* [129] of floats. Currently, all models in JAJAPY use an explicit state-space representation.

The `Base_MC` and `Parametric_Model` classes have an attribute `labelling` containing the label associated to each state. This attribute is a Python *list* whose length equals the number of states of the model.

Such an attribute does not exist for `Base_HMM`, since its states are not associated with one label but a probability distribution. The `HMM` class has an attribute `output`, a *Numpy ndarray* or a Python *list* of dimension $|S| \times |\mathcal{L}|$, describing this probability distribution. The `output` attribute of the `GoHMM` class is a *Numpy ndarray* of dimension $|S| \times n \times 2$, with n the degree of the GoHMM (i.e. the number of Gaussian distributions associated to each state) containing the two parameters μ and σ for each distribution.

Finally, JAJAPY uses Sympy [130] to represent symbolic expressions used for transition rate expressions in parametric models: (i) the `transition_expr` attribute is a Python *list* of *str* containing the different polynomial functional used by the model transitions. (ii) The `matrix` attribute contains $|S|^2$ integers corresponding to `transition_expr` indexes. For instance, `matrix[s1,s2] == 3` means that the transition from state `s1` to state `s2` is expressed by `transition_expr[3]`.

(iii) The `parameter_str` attribute is a Python *list* of *str* containing the name of all parameters, (iv) the `parameter_values` attribute is a Python *dict* where the keys are *str* corresponding to parameter names, and the value is a *float* corresponding to the value of the parameter (or a *numpy.nan* if the parameter is not instantiated), and (v) `parameter_indexes` is a *list* of *Numpy ndarray* containing, for each parameters, the transitions in which this parameter is used in the expression. Given a transition expression (from `transition_expr`) and the `parameter_values` dictionary, the Sympy library evaluates the value of this expression, assuming that all the parameters used in the expression are associated to a numerical value in the dictionary.

Learning with BW. BW executions are handled by the `BW` class.

The `BW.fit` method starts by determining which model formalism should be used according to the given initial model (if provided) and the training set. Then, it selects the appropriate update procedure and runs the BW algorithm.

JAJAPY aims at reducing the number of computation. An execution of the BW algorithm resolves into a sequence of matrix operations that are handled by Numpy. In addition, if JAJAPY is executed on a Linux machine, it supports multithreading to speed up the BW algorithm: at each BW iteration, JAJAPY executes one thread for each unique trace in the training set. Otherwise, each unique trace is processed sequentially.

Output models. The output format of any JAJAPY learning methods can be chosen among the following: STORMPY sparse model or JAJAPY model. The output model can also be exported to a PRISM file by setting the `output_file_prism` parameter of the `BW.fit` method.

Representing training sets and test sets. JAJAPY uses its own `Set` class to represent training and test sets. This class has two attributes: (i) `sequences`, the set of all unique traces in the training set, and (ii) `times`, which contains, for each trace in `sequences`, the number of times this trace has been observed. This reduces significantly the number of computations during the learning process when traces appear several times in the training set. Nevertheless, the training set can be given as a Python list or a *Numpy ndarray* to the `BW.fit` method.

In JAJAPY training sets and test sets are not represented through a prefix tree (as is normal in other libraries) since this is only advantageous when JAJAPY is used in single-thread mode. The training sets/test sets are sorted by JAJAPY only in this case.

7.5 Experimental evaluation and comparison

In this Section, we first test JAJAPY validity. Secondly, we empirically evaluate how the different learning methods scale with the size of the output model and the training set. Finally, we compare it with AALPY.

All the experiments were run on a Linux machine with an AMD Ryzen 9 3900X 12-Core processor and 32 GB of memory.

7.5.1 JAJAPY validation testing

We test JAJAPY validity as follows: (i) we translate a STORMPY model \mathcal{M} representing the Yao-Knuth’s die [3] to a JAJAPY one; (ii) we use it to generate a training set of 10,000 traces of length 10: 10 being big enough to reach the final state with a decent probability and 10,000 being small enough to learn the model in few seconds, but sufficiently big to learn a correct approximation of the SUL; (iii) we learn, using JAJAPY BW and Alergia implementations, two new STORMPY models \mathcal{M}' and \mathcal{M}'' , and finally (iv) \mathcal{M}' , \mathcal{M}'' are compared both w.r.t. their outcomes on some relevant model checking queries and their loglikelihood distance on a test set relative to the true model \mathcal{M} .

The first three queries correspond to the probability that the die roll gives us 1, 2 or 3. The next three queries indicate the probability that the die gives us 4, 5 or 6 without ever going through the same state (except the final one) more than once. Finally the last query corresponds to the probability that 10 throws of the coin are enough to simulate the roll of the die.

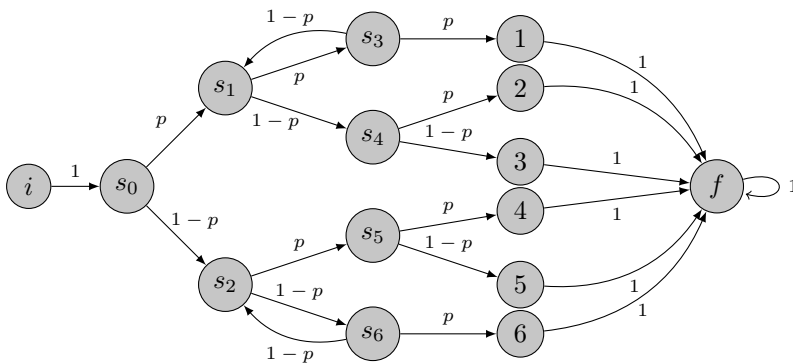


Figure 7.4 – The Yao-Knuth’s die from [3]

We run these experiments on a Markov chain modelling the Yao-Knuth’s die represented in Figure 7.4 once with $p = 0.5$ (i.e. with a unbiased coin) and once with $p = 0.9$. Table 7.2 and 7.3 show that JAJAPY learned a valid representation of the source model regardless of the algorithm used. When the coin is unbiased, Alergia learns a bigger model than BW (23 states against 14) which is better in terms of loglikelihood distance but worst for the model checking queries. This is explained by the fact that, in this case, Alergia is not able to merge a large number of states and, therefore, generates a model close to a PTA, which is efficient in terms of loglikelihood distance (especially when the sequences in the test set are the same length as those in the training set). When the coin is biased, some possible traces do not appear or appear very little in the training set. Therefore, the training set being composed of much more similar traces, the initial PTA is much smaller as

	true	BW	Alergia
# states	14	14	23
$Pr(F(1))$	0.167	0.168	0.168
$Pr(F(2))$	0.167	0.170	0.169
$Pr(F(3))$	0.167	0.163	0.163
$Pr(F^{\leq 4}(4))$	0.125	0.130	0.143
$Pr(F^{\leq 4}(5))$	0.125	0.124	0.136
$Pr(F^{\leq 4}(6))$	0.125	0.107	0.129
$Pr(F^{\leq 10}(f))$	0.996	0.979	0.973
ll. distance	0.0	1.700	1.616
learning time (s)	-	1.039	0.003

Table 7.2 – Results for an unbiased Yao-Knuth’s die ($p = 0.5$).

	true	BW	Alergia
# states	14	14	12
$Pr(F(1))$	0.801	0.797	0.797
$Pr(F(2))$	0.089	0.092	0.092
$Pr(F(3))$	0.010	0.008	0.008
$Pr(F^{\leq 4}(4))$	0.081	0.088	0.076
$Pr(F^{\leq 4}(5))$	0.009	0.010	0.009
$Pr(F^{\leq 4}(6))$	0.001	0.002	0.002
$Pr(F^{\leq 10}(f))$	1.0	0.999	0.992
ll. distance	0.0	0.511	1.569
learning time (s)	-	1.060	0.001

Table 7.3 – Results for a biased Yao-Knuth’s die ($p = 0.9$).

well as the model generated by Alergia. On the other hand, the likelihood of the sequences present in the test set and not in the training set can be fairly different between the model generated by Alergia and the SUL. In other words, for the same training set, a model generated by Alergia will often be less general than one generated by BW, because Alergia is more sensitive to overfitting.

7.5.2 Experimental evaluation of the scalability

Scalability evaluation for MCs, CTMCs and MDPs.

To evaluate the scalability of our software, we report the running time and the memory footprint required to learn models with an increasing number of states.

We use JAJAPY to learn randomly generated transition-labeled MCs and CTMCs ranging from 10 to 200 states, corresponding to models with up to 100 to 40,000 parameters (the number of parameters is at most s^2 , where s is the number of states). We perform the same experiment for MDPs with 5 to 100 states and 4 actions, thus having at most 100 to 40,000 parameters (here the number of parameters is at most $s^2 \cdot a$, where s and a are respectively the number of states and actions). We employ training sets containing 1,000 traces of length 10. These two values offer a good compromise between accuracy and running time. We set the size of the initial hypothesis equal to that of the SUL. The results are shown in Fig. 7.5.

The running time for all type of SULs increases exponentially, but at a larger rate for CTMCs: while one BW iteration for an MDP with 200 states and an MC with 400 states took around two minutes in this setting, one BW iteration for a CTMC with 200 states took 97 minutes. This is due to the computational difficulty of calculating rates of exponential distributions when learning CTMCs parameters. Memory usage also increases exponentially for all types of Markov models. These

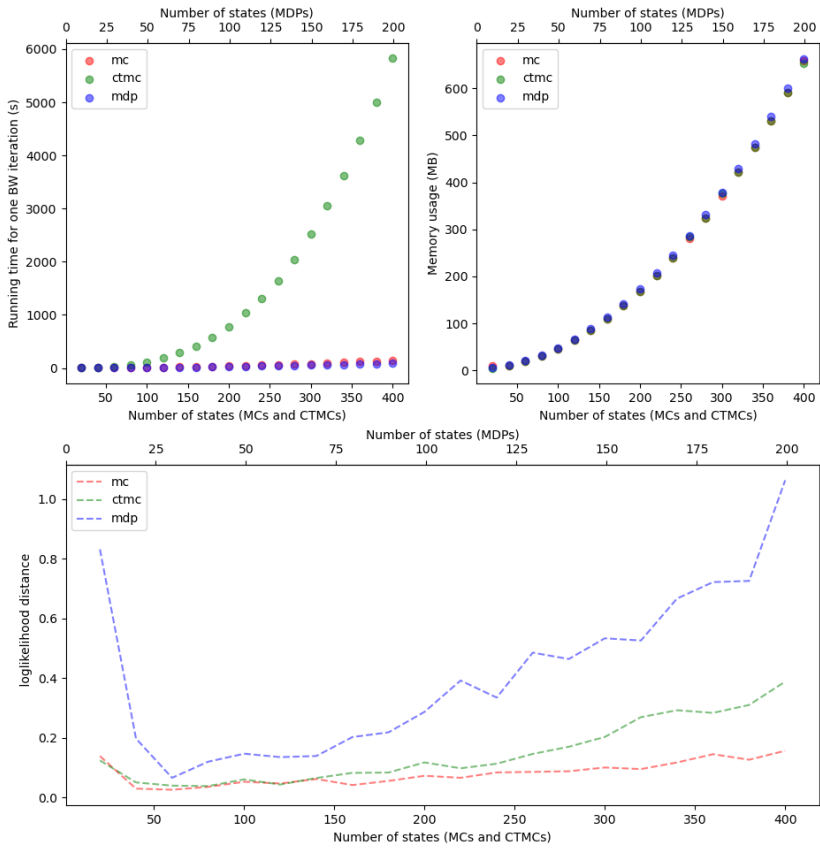


Figure 7.5 – JAJAPY running time, memory usage and loglikelihood distance w.r.t. the number of parameters of the hypothesis.

exponential growths were expected, since the number of parameters to estimate increases exponentially with the number of states.

Finally, as the complexity of the model increases, the loglikelihood distance grows. This issue is usually mitigated by increasing the length and number of traces in the training set. Another experiment, presented in Appendix C, demonstrates that doubling the number of traces in the training set reduces the loglikelihood distance by half.

Scalability evaluation for pCTMCs.

To evaluate the scalability of our software on pCTMCs, we refer to tandem queuing experiment in Chapter 6, Section 6.5.

7.5.3 Comparison with HMMLEARN

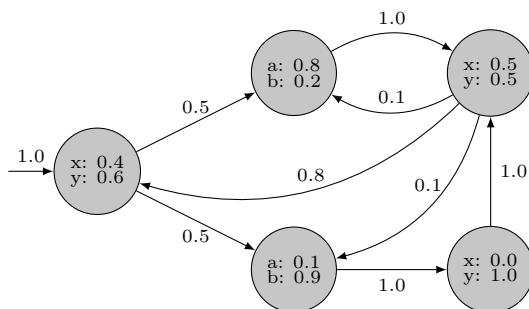


Figure 7.6 – An HMM with 5 states.

In Chapter 4, we introduced HMMLEARN, a Python library that implements the BW algorithm for various extensions of HMMs, including GoHMM and HMM itself. In the following, we compare the performance of both JAJAPY and HMMLEARN in terms of running time and log-likelihood distance while learning the HMM depicted in Figure 7.6.

For this experiment, both the training and test sets comprise 1,000 traces, each with a length of 10. Both JAJAPY and HMMLEARN execute 20 BW iterations using the same training set and commencing with identical 5-state hypotheses. This experiment is repeated 10 times, with variations in the initial hypotheses. The detailed results are presented in Table 7.4.

	HMMLEARN	JAJAPY
average ll. distance	0.714	0.416
average running time (s)	1.299	9.044

Table 7.4 – Results for the comparison of JAJAPY and HMMLEARN.

Algorithm	Model	JAJAPY	AALPY
BW-MC	MC	✓	✗
BW-CTMC	CTMC	✓	✗
BW-HMM	HMM	✓	✗
BW-GoHMM	GoHMM	✓	✗
MM-pCTMC	pCTMC	✓	✗
BW-MDP	MDP	✓	✗
Active-BW	MDP	✓	✗
Alergia	MC	✓	✓
IOAlergia	MDP	✓	✓
L_{MDP}^*	MDP	✗	✓

Table 7.5 – Learning algorithms for Markov models implemented in JAJAPY and AALPY.

The results suggest that while JAJAPY is comparatively slower than HMMLEARN, it demonstrates an ability to learn more accurate approximations in terms of log-likelihood distance.

HMMLEARN is faster than JAJAPY because it runs compiled C code. However, it's important to note that HMMLEARN only partially updates the transition probabilities during each Baum-Welch (BW) iteration. Let $b_{s,s'}$ represent the prior probability of a transition from state s to s' , and $b'_{s,s'}$ represent the new probability as computed by the BW algorithm. In HMMLEARN, this transition probability is updated to:

$$\frac{b_{s,s'} + b'_{s,s'} - 1}{\sum_{s'' \in S} b_{s,s''} + b'_{s,s''} - 1}$$

As a result of this update mechanism, HMMLEARN requires more iterations to achieve the same level of accuracy as JAJAPY .

7.5.4 Comparison with AALPY

AALPY is an active automata learning python library. AALPY implements several learning algorithms to learn various families of automata. Since JAJAPY learns stochastic models only, we will focus here on these models. However, AALPY is also able to learn non-stochastic models, as Deterministic Finite Automata or Mealy Machines.

AALPY implements L_{MDP}^* , an extension of Angluin's L^* algorithm [67, 33] to learn MDPs, and the Alergia algorithm to learn MCs.

Table 7.5 summarises all the learning algorithms available in JAJAPY and AALPY for stochastic models. We can notice that JAJAPY can learn non-deterministic models and CTMCs, which AALPY cannot. On the other hand, AALPY can learn non-stochastic models, which JAJAPY cannot. Finally, in contrast to AALPY, JAJAPY output models are immediately usable in Stormpy.

JAJAPY Alergia vs AALPY Alergia

First, we compare Alergia implementations by JAJAPY and AALPY for two Yao-Kunth'dice (one with $p = 0.5$ and one with $p = 0.9$), using two training sets containing each 10,000 traces of length 10. The results are summarised in Table 7.6 and 7.7.

There is no difference in the results obtained between these two implementations.

Formula	Original	JAJAPY	AALPY
# states	14	14	14
$Pr(F(1))$	0.167	0.164	0.164
$Pr(F(2))$	0.167	0.167	0.167
$Pr(F(3))$	0.167	0.165	0.165
$Pr(F^{\leq 4}(3))$	0.125	0.126	0.126
$Pr(F^{\leq 4}(3))$	0.125	0.125	0.125
$Pr(F^{\leq 4}(3))$	0.125	0.126	0.126
$Pr(F^{\leq 10}(f))$	0.996	0.996	0.996
ll distance	0.0	0.0	0.0
learning time	-	0.001	0.026

Table 7.6 – Results for an unbiased Yao-Knuth's die ($p = 0.5$).

Formula	Original	JAJAPY	AALPY
# states	14	14	14
$Pr(F(1))$	0.801	0.807	0.807
$Pr(F(2))$	0.089	0.088	0.088
$Pr(F(3))$	0.01	0.01	0.01
$Pr(F^{\leq 4}(3))$	0.081	0.076	0.076
$Pr(F^{\leq 4}(3))$	0.009	0.007	0.007
$Pr(F^{\leq 4}(3))$	0.001	0.001	0.001
$Pr(F^{\leq 10}(f))$	1.0	1.0	1.0
ll distance	0.0	0.0	0.0
learning time	-	0.001	0.027

Table 7.7 – Results for a biased Yao-Knuth's die ($p = 0.9$).

Only the execution time differs: JAJAPY being faster than AALPY.

JAJAPY BW vs AALPY L_{MDP}^*

We compare AALPY L^* and JAJAPY BW algorithms on learning two variants of the grid-worlds presented in [71] and illustrated in Figure 7.7. this experiment may seem similar to the one in Chapter 5, section 5.4, but it is not. In fact, here we compare the L_{MDP}^* implementation offered by AALPY against the classic BW implementation offered by JAJAPY , whereas in Chapter 5, we were comparing it with Active-BW.



Figure 7.7 – Grid worlds models. (Left) a 3x3 deterministic model; (Right) a 4x4 non-deterministic model.

	true	AALPY	JAJAPY BW
overall # of labels	-	74,285	74,285
# of traces	-	15,218	3,714
# of states	17	18	17
loglikelihood distance	0.0	0.7305	0.3352
$Pr_{\max}(F^{\leq 4}(\text{goal}))$	0.336	0.322	0.347
$Pr_{\max}(\neg G U^{\leq 4}(\text{goal}))$	0.072	0.074	0.074
Running time	-	1.15 s	290.8 s

Table 7.8 – Results for the 3x3 deterministic grid-world model.

	true	AALPY	JAJAPY BW
overall # of labels	-	16,232,244	200,000
# of traces	-	2,174,167	10,000
# of states	28	207	28
loglikelihood distance	0.0	0.4963	0.4680
$Pr_{\max}(F^{\leq 7}(\text{goal}))$	0.687	0.680	0.692
$Pr_{\max}(F^{\leq 12}(\text{goal}))$	0.996	0.995	0.996
$Pr_{\max}(\neg(C W) U^{\leq 7}(\text{goal}))$	0.520	0.514	0.504
Running time	-	290.65 s	15,303.83 s

Table 7.9 – Results for the 4x4 non-deterministic grid-world model.

We run, for both models, AALPY for 200 L^* learning iterations and JAJAPY for 200 BW iterations. We emphasize the fact that the two tools are using two different learning algorithms that are, in the author’s opinion, complementary.

Table 7.8 and 7.9 show the results respectively for the 3x3 grid and the 4x4 grid. In both cases, the loglikelihood distance is computed for a test set containing 10,000 traces of length 20. First, we observe that, when the SUL is deterministic, the two output models are similar. Actually, JAJAPY output is slightly closer to the SUL, but AALPY ran faster. However, when the SUL is non-deterministic, the difference between the two output models is more important. AALPY ran faster but produced a model with almost 8 times more states. Indeed L_{MDP}^* , by property, learned a deterministic approximation of the SUL, that is much bigger than the SUL itself. In terms of loglikelihood distance, AALPY output model is slightly less accurate than JAJAPY one. Finally, we notice that JAJAPY uses far less information than AALPY, and does not require any interaction with the SUL (using a passive learning approach), in contrast to AALPY.

The fact that AALPY runs faster than JAJAPY can be explained by the complexity of the two algorithms involved here, namely L_{MDP}^* and the BW algorithm. The BW algorithm is known to be costly in terms of time and memory complexity. Khreich et al. [41] point out several cases where, due to its cost, the BW algorithm could not be applied. In the same paper, they present a variant of it, requiring fewer memory resources while achieving the same results. However, Bartolucci et

al. [131] show that this variant suffers from numerical problems.

In general, when learning MDPs, if it is impossible to interact with the SUL, we recommend JAJAPY BW. Otherwise, we recommend using AALPY L_{MDP}^* , especially when the SUL is known to be deterministic.

7.6 Conclusions and Future Work

We presented JAJAPY, a Python learning library for Markov models, and discussed its key features, implementation, usage, and performance evaluation. JAJAPY is designed to be interoperable with PRISM and STORM, and offers a variety of learning methods, both active and passive. We compared JAJAPY and AALPY and argued that the two libraries complement each other, thus the choice of which library to use depends on the learning scenario.

As a future work, we consider implementing GPU-accelerated methods to speed-up the forward-backward computations required at each iteration of the BW algorithms borrowing ideas from [102, 132].

Part III

Epilogue

Chapter 8

Closing Remarks

In this thesis, we studied the problem of modelling Markov models from empirical (partial) observations of the system under learning. We tackled this problem from different angles both considering different types of Markov models and different sampling scenarios. The technical contributions of this thesis are

1. an active learning strategy to enhance the accuracy of the Baum-Welch algorithm to learn MDPs. We showed that this algorithm overperformed the classic BW algorithm, but requires the learner to interact with the SUL.
2. A parameter estimation procedure for parametric continuous-time Markov chains expressed in the PRISM language. The algorithm has been developed following the iterative optimisation framework known as MM algorithm. Notably, the estimation procedure imposes little requirements on the input training data. Indeed it is robust to missing dwell time data and assumes that states are only observable through their atomic propositions.
3. An open source Python library implementing various learning algorithms spanning different families of Markov models. The library is designed to be compatible with PRISM and interoperable with STORM via STORMPY. This makes JAJAPY a valuable aid in the modelling process of safety-critical systems as well as a useful tool for model-driven development.

By expanding upon existing methodologies, we have enriched the field of learning stochastic models by devising two novel algorithms: one improving the performance of the BW algorithm to learn MDPs (assuming that the learner can interact with the SUL), and the second by learning synchronous compositions of CTMCs from traces with missing dwell times, enhancing the applicability of Markov models to real-world scenarios where data might be incomplete.

Additionally, our contributions extend to the practical realm of system analysis and verification. We have augmented the capabilities of existing model checkers STORM and PRISM by developing an open-source Python library, JAJAPY, compatible with both of them. This integration bridges the gap between model learning and formal verification, enabling practitioners to seamlessly transition from the modelling phase to comprehensive system analysis. By harmonising learning algorithms with model checking techniques, our approach facilitates the identification of potential

issues and vulnerabilities within critical systems.

It is worth mentioning that JAJAPY was successfully employed in the research conducted within the ongoing ‘Sleep Revolution’ project at the University of Reykjavik. One goal of this project is to automatically assess the sleep quality of specific patients using recorded physiological signals such as EEG and PSG, aiming to detect sleep disorders. Specifically, we employed JAJAPY BW implementation to learn a GoHMM, aimed at pinpointing the grey zones, i.e., areas where our alternate estimator (an autoencoder) struggles to accurately identify sleep stages. As this work is still in progress, we will refrain from delving deeper into the specifics here.

In summary, this thesis represents a multifaceted advancement in the domain of learning stochastic models. Through the development of innovative algorithms, open-source libraries, and integrative approaches, we have contributed to the broader fields of machine learning, system analysis, and verification. These achievements collectively pave the way for enhanced modelling accuracy, refined parameter estimation, and more rigorous assessments of critical systems in real-world applications.

Appendix A

The Baum-Welch algorithm in details

A.1 Convergence of the EM algorithm

Recall that the point of the EM algorithm is to find θ such that, for of observed data X and a set of latent data Z , it maximises

$$l(\theta; X) = Pr^\theta(X) = \sum_z Pr^\theta(z) Pr^\theta(X | z) \quad (\text{A.1})$$

The above likelihood function is often intractable since z is unobserved and its distribution $Pr(z; \theta)$ is typically determined by the parameters θ .

The EM algorithm is an iterative optimisation technique aimed at maximising Equation (A.1). Starting from an initial parameter estimate θ_0 , the current parameter estimate θ_m is updated by applying the following steps:

Expectation step (E-step) Compute the expected value of the log-likelihood function relative to the conditional distribution of Z , given the observed data X and the current parameter value estimates θ_m

$$Q(\theta|\theta_m) = \sum_z Pr^{\theta_m}(z | X) \ln Pr^\theta(X, z) \quad (\text{A.2})$$

Maximisation step (M-step) The next parameter estimates θ_{m+1} are found as those achieving the maximum value of $Q(\theta|\theta_m)$. Formally

$$\theta_{m+1} = \operatorname{argmax}_\theta Q(\theta|\theta_m).$$

Notably, the new parameter estimate improves with respect to the previous one in the sense that $l(\theta_m; X) \leq l(\theta_{m+1}; X)$.

Theorem (Convergence of the EM algorithm). *The EM algorithm converges to a local maximum of the likelihood of the training set [38].*

Proof. The EM algorithm is an iterative procedure for maximising $l(\theta; X)$, equivalently $\ln l(\theta; X)$. Assume that after the m^{th} iteration the current estimate for θ is given by θ_m . Since the objective is to maximise $\ln l(\theta; X)$, we want to maximise the difference

$$\begin{aligned}
& \ln l(\theta; X) - \ln(\theta_m; X) \\
&= \ln Pr^\theta(X) - \ln Pr^{\theta_m}(X) \\
&= \ln \sum_z Pr^\theta(z) Pr^\theta(X | z) - \ln Pr^{\theta_m}(X) \\
&= \ln \sum_z Pr^\theta(z) Pr^\theta(X | z) \frac{Pr^{\theta_m}(z | X)}{Pr^{\theta_m}(z | X)} - \ln Pr^{\theta_m}(X) \\
&= \ln \sum_z Pr^{\theta_m}(z | X) \frac{Pr^\theta(z) Pr^\theta(X | z)}{Pr^{\theta_m}(z | X)} - \ln Pr^{\theta_m}(X)
\end{aligned}$$

By Jensen's inequality :

$$\begin{aligned}
& \geq \sum_z Pr^{\theta_m}(z | X) \ln \frac{Pr^\theta(z) Pr^\theta(X | z)}{Pr^{\theta_m}(z | X)} - \ln Pr^{\theta_m}(X) \\
&= \sum_z Pr^{\theta_m}(z | X) \ln \frac{Pr^\theta(z) Pr^\theta(X | z)}{Pr^{\theta_m}(z | X) Pr^{\theta_m}(X)} \\
&:= \Delta(\theta | \theta_m).
\end{aligned}$$

Therefore:

$$\ln l(\theta; X) \geq \ln(\theta_m; X) + \Delta(\theta | \theta_m).$$

We define

$$\mathcal{L}(\theta | \theta_m) = \ln(\theta_m; X) + \Delta(\theta | \theta_m).$$

Then

$$\ln l(\theta; X) \geq \mathcal{L}(\theta | \theta_m)$$

Additionally, we observe that $\Delta(\theta_m | \theta_m) = 0$, and consequently $\mathcal{L}(\theta_m | \theta_m) = \ln l(\theta_m; X)$.

Thus, $\mathcal{L}(\theta | \theta_m)$ is bounded above by $\ln l(\theta; X)$ and, for $\theta = \theta_m$, $\mathcal{L}(\theta | \theta_m) = \ln l(\theta; X)$. Therefore, by selecting θ maximising $\mathcal{L}(\theta | \theta_m)$, we would achieve the greatest possible increase in $\ln l(\theta; X)$.

$$\begin{aligned}
\theta_{m+1} &= \arg \max_{\theta} \mathcal{L}(\theta | \theta_m) \\
&= \arg \max_{\theta} \ln(\theta_m; X) + \sum_z Pr^{\theta_m}(z | X) \ln \frac{Pr^\theta(z) Pr^\theta(X | z)}{Pr^{\theta_m}(z | X) Pr^{\theta_m}(X)} \\
&= \arg \max_{\theta} \sum_z Pr^{\theta_m}(z | X) \ln Pr^\theta(z) Pr^\theta(X | z) \\
&= \arg \max_{\theta} \sum_z Pr^{\theta_m}(z | X) \ln Pr^\theta(X, z) \\
&= \arg \max_{\theta} Q(\theta | \theta_m)
\end{aligned}$$

□

A.2 Baum-Welch for MCs

The forward-backward functions For $o = \ell_0 \dots \ell_T$ a trace and an MC \mathcal{M} , we define the forward and the backward functions $\alpha_o, \beta_o: S \times \{0 \dots T\} \rightarrow [0, 1]$ as

$$\begin{aligned}\alpha_o(s, i) &= Pr^{\mathcal{M}}[Y_{0:i} = \ell_0 \dots \ell_i, X_i = s], \text{ and} \\ \beta_o(s, i) &= Pr^{\mathcal{M}}[Y_{i:T} = \ell_i \dots \ell_T | X_i = s].\end{aligned}$$

These can be calculated according to the following recurrences

$$\begin{aligned}\alpha_o(s, i) &= \begin{cases} 1_{\ell_0}(\ell(s)) \cdot \pi(s) & \text{if } i = 0 \\ 1_{\ell_i}(\ell(s)) \cdot \sum_{s' \in S} \alpha_o(s', i-1) \cdot \tau(s')(s) & \text{if } 0 < i \leq T \end{cases} \\ \beta_o(s, i) &= \begin{cases} 1_{\ell_T}(\ell(s)) & \text{if } i = T \\ 1_{\ell_i}(\ell(s)) \cdot \sum_{s' \in S} \tau(s)(s') \cdot \beta_o(s', i+1) & \text{if } 0 \leq i < T \end{cases}\end{aligned}$$

Thus:

$$\begin{aligned}\gamma_o(s, i) &= \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)} \\ \xi_o(s, i)(s') &= \frac{\alpha_o(s, i) \cdot \tau(s)(s') \cdot \beta_o(s', i+1)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}\end{aligned}$$

Additionally, we have:

$$\ln l(\mathcal{M}; \rho) = \ln \pi(s_0) + \sum_{i=0}^{|\rho|-1} \ln \tau(s_i)(s_{i+1}) \quad (\text{A.3})$$

The E and M steps The EM algorithm can be described as the following E and M steps repeated until convergence:

1. (E Step) Compute $Q(\theta | \theta_m) = \sum_z Pr^{\theta_m}(z | X) \ln Pr^\theta(X, z)$,
2. (M Step) Set $\theta_{m+1} = \arg \max_{\theta} Q(\theta | \theta_m)$,

where X in the set of observed data and θ_m the current estimated parameters.

In the context of learning an MC from a given finite set \mathcal{O} of traces, the E and M steps are:

1. (E Step) Compute $Q(\mathcal{M} | \mathcal{M}_m) = \sum_{o \in \mathcal{O}} \sum_{\rho \in \text{Paths}(o)} \ln [l(\mathcal{M}; \rho)] l(\mathcal{M}_m; \rho)$.
2. (M Step) Set $\mathcal{M}_{m+1} = \arg \max_{\mathcal{M}} Q(\mathcal{M} | \mathcal{M}_m)$.

Let $\mathcal{M}_m = \langle S, \mathcal{L}, \ell, \tau, \pi \rangle$ and $\mathcal{M} = \langle S, \mathcal{L}, \ell, \hat{\tau}, \hat{\pi} \rangle$.

First, plugging (A.3) into $Q(\mathcal{M} \mid \mathcal{M}_m)$, we get:

$$\begin{aligned} Q(\mathcal{M} \mid \mathcal{M}_m) &= \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \ln \hat{\pi}(s_0) \cdot l(\mathcal{M}_m; \rho) \\ &\quad + \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \sum_{i=0}^{|\rho|-1} \ln \hat{\tau}(s_i)(s_{i+1}) \cdot l(\mathcal{M}_m; \rho) \end{aligned}$$

Now we optimise with Lagrange multipliers (l_π and l_{τ_s}). Let $L(\mathcal{M}, \mathcal{M}_m)$ be the Lagrangian:

$$L(\mathcal{M}, \mathcal{M}_m) = Q(\mathcal{M} \mid \mathcal{M}_m) - l_\pi \left(\sum_{s' \in S} \hat{\pi}(s') - 1 \right) - \sum_{s \in S} l_{\tau_s} \left(\sum_{s' \in S} \hat{\tau}(s)(s') - 1 \right)$$

Estimation of π First, we focus on the initial state distribution π :

$$\begin{aligned} \frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial \hat{\pi}_s} &= \frac{\partial Q(\mathcal{M} \mid \mathcal{M}_m)}{\partial \hat{\pi}_s} - l_\pi = 0 \\ &= \frac{\partial}{\partial \hat{\pi}_s} \left(\sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \ln \hat{\pi}(s_0) l(\mathcal{M}_m; \rho) \right) - l_\pi = 0 \\ &= \frac{\partial}{\partial \hat{\pi}_s} \left(\sum_{o \in \mathcal{O}} \sum_{s' \in S} \ln \hat{\pi}(s') Pr^{\mathcal{M}_m}(X_0 = s', O_{|o|} = o) \right) - l_\pi = 0 \\ &= \sum_{o \in \mathcal{O}} \frac{Pr^{\mathcal{M}_m}(X_0 = s, O_{|o|} = o)}{\hat{\pi}_s} - l_\pi = 0 \end{aligned}$$

Hence:

$$\hat{\pi}_s = \sum_{o \in \mathcal{O}} \frac{Pr^{\mathcal{M}_m}(X_0 = s, O_{|o|} = o)}{l_\pi} \quad (\text{A.4})$$

Furthermore:

$$\frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial l_\pi} = - \left(\sum_{s' \in S} \hat{\pi}(s') - 1 \right) = 0 \quad (\text{A.5})$$

By plugging (A.4) into (A.5) we get:

$$l_\pi = \sum_{o \in \mathcal{O}} \sum_{s' \in S} Pr^{\mathcal{M}_m}(X_0 = s', O_{|o|} = o) \quad (\text{A.6})$$

And by plugging (A.6) into (A.4):

$$\begin{aligned} \hat{\pi}_s &= \frac{\sum_{o \in \mathcal{O}} Pr^{\mathcal{M}_m}(X_0 = s, O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{s' \in S} Pr^{\mathcal{M}_m}(X_0 = s', O_{|o|} = o)} \\ \hat{\pi}_s &= \frac{\sum_{o \in \mathcal{O}} Pr^{\mathcal{M}_m}(X_0 = s \mid O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{s' \in S} Pr^{\mathcal{M}_m}(X_0 = s' \mid O_{|o|} = o)} \end{aligned}$$

Finally, using the previously defined coefficients:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{s' \in \mathcal{S}} \gamma_o(s', 0)}$$

Estimation of τ Now, we focus on the transition probability distributions τ :

$$\begin{aligned} \frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial \hat{\tau}_{s,s'}} &= \frac{\partial}{\partial \hat{\tau}_{s,s'}} \left(\sum_{o \in \mathcal{O}} \sum_{\rho \in \text{Paths}(o)} \sum_{i=0}^{|\rho|-1} \ln[\hat{\tau}(s_i)(s_{i+1})] l(\mathcal{M}_m; \rho) \right) - l_{\tau_s} = 0 \\ &= \frac{\partial}{\partial \hat{\tau}_{s,s'}} \left(\sum_{o \in \mathcal{O}} \sum_{u, u' \in \mathcal{S}} \sum_{i=0}^{|\rho|-1} \ln[\hat{\tau}_{u,u'}] P_{r^{\mathcal{M}_m}}(X_i = u, X_{i+1} = u', O_{|o|} = o) \right) - l_{\tau_s} = 0 \\ &= \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} \frac{P_{r^{\mathcal{M}_m}}(X_i = s, X_{i+1} = s', O_{|o|} = o)}{\hat{\tau}_{s,s'}} - l_{\tau_s} = 0 \end{aligned}$$

Hence:

$$\hat{\tau}_{s,s'} = \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} \frac{P_{r^{\mathcal{M}_m}}(X_i = s, X_{i+1} = s', O_{|o|} = o)}{l_{\tau_s}} \quad (\text{A.7})$$

Furthermore:

$$\frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial l_{\tau_s}} = - \left(\sum_{s' \in \mathcal{S}} \hat{\tau}(s)(s') - 1 \right) = 0 \quad (\text{A.8})$$

By plugging (A.7) into (A.8) we get:

$$l_{\tau_s} = \sum_{o \in \mathcal{O}} \sum_{s' \in \mathcal{S}} \sum_{i=0}^{|\rho|-1} P_{r^{\mathcal{M}_m}}(X_i = s, X_{i+1} = s', O_{|o|} = o) \quad (\text{A.9})$$

$$= \sum_{o \in \mathcal{O}} \sum_{i=1}^{|\rho|-1} P_{r^{\mathcal{M}_m}}(X_i = s, O_{|o|} = o) \quad (\text{A.10})$$

And by plugging (A.10) into (A.7):

$$\begin{aligned} \hat{\tau}_{s,s'} &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} P_{r^{\mathcal{M}_m}}(X_i = s, X_{i+1} = s', O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} P_{r^{\mathcal{M}_m}}(X_i = s, O_{|o|} = o)} \\ \hat{\tau}_{s,s'} &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} P_{r^{\mathcal{M}_m}}(X_i = s, X_{i+1} = s' \mid O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} P_{r^{\mathcal{M}_m}}(X_i = s \mid O_{|o|} = o)} \end{aligned}$$

Finally, using the previously defined coefficients:

$$\hat{\tau}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=1}^{|\rho|-1} \gamma_o(s, i)}$$

Conclusion For MCs, the UPDATE procedure updates π and τ as follows:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{u \in \mathcal{S}} \gamma_o(u, 0)}$$

$$\hat{\tau}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=1}^{|\rho|-1} \gamma_o(s, i)}$$

Remark A.2.1. One may incur in the situation where $\sum_{o \in \mathcal{O}} \sum_{i=1}^{|\rho|-1} \gamma_o(s, i) = 0$, indicating that the state s does not play a role in the observed dynamics. In this case the update procedure leaves the distribution $\tau(s)$ unchanged.

A.3 Baum-Welch for MDPs

This Appendix Section is close to the previous one, the only difference being the input actions.

The forward-backward functions For $o = (\ell_0, a_0) \dots \ell_T$ a trace and an MDP \mathcal{M} , we define the forward and the backward functions $\alpha_o, \beta_o: \mathcal{S} \times \{0 \dots T\} \rightarrow [0, 1]$ as

$$\alpha_o(s, i) = Pr^{\mathcal{M}}[Y_{0:i} = \ell_0 \dots \ell_i, X_i = s \mid A_{0:i-1} = a_0 \dots a_{i-1}], \text{ and}$$

$$\beta_o(s, i) = Pr^{\mathcal{M}}[Y_{i:T} = \ell_i \dots \ell_T \mid A_{i:T-1} = a_i \dots a_{T-1}, X_i = s].$$

These can be calculated according to the following recurrences

$$\alpha_o(s, i) = \begin{cases} 1_{\ell_0}(\ell(s)) \cdot \pi(s) & \text{if } i = 0 \\ 1_{\ell_i}(\ell(s)) \cdot \sum_{s' \in \mathcal{S}} \alpha_o(s', i-1) \cdot \tau_{a_{i-1}}(s')(s) & \text{if } 0 < i \leq T \end{cases}$$

$$\beta_o(s, i) = \begin{cases} 1_{\ell_T}(\ell(s)) & \text{if } i = T \\ 1_{\ell_i}(\ell(s)) \cdot \sum_{s' \in \mathcal{S}} \tau_{a_i}(s)(s') \cdot \beta_o(s', i+1) & \text{if } 0 \leq i < T \end{cases}$$

Thus:

$$\gamma_o(s, i) = \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{u \in \mathcal{S}} \alpha_o(u, i) \beta_o(u, i)}$$

$$\xi_o(s, i)(s') = \frac{\alpha_o(s, i) \cdot \tau_{a_i}(s)(s') \cdot \beta_o(s', i+1)}{\sum_{u \in \mathcal{S}} \alpha_o(u, i) \beta_o(u, i)}$$

Additionally, we have:

$$\ln l(\mathcal{M}; \rho) = \ln \pi(s_0) + \sum_{i=0}^{|\rho|-1} \ln \tau_{a_i}(s_i)(s_{i+1}) \quad (\text{A.11})$$

The Lagrangian multipliers On a given finite set \mathcal{O} of traces, the Baum-Welch algorithm can be described as repeating the two following steps until convergence:

1. (E step) Compute $Q(\mathcal{M} \mid \mathcal{M}_m) = \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \ln [l(\mathcal{M}; \rho)] l(\mathcal{M}_m; \rho)$.
2. (M step) Set $\mathcal{M}_{m+1} = \arg \max_{\mathcal{M}} Q(\mathcal{M} \mid \mathcal{M}_m)$.

Let $\mathcal{M}_m = \langle S, \mathcal{L}, \ell, A, \{\tau_a\}_{a \in A}, \pi \rangle$ and $\mathcal{M} = \langle S, \mathcal{L}, \ell, A, \{\hat{\tau}_a\}_{a \in A}, \hat{\pi} \rangle$. First, plugging (A.11) into $Q(\mathcal{M} \mid \mathcal{M}_m)$, we get:

$$\begin{aligned} Q(\mathcal{M} \mid \mathcal{M}_m) &= \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \ln \hat{\pi}(s_0) \cdot l(\mathcal{M}_m; \rho) \\ &+ \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \sum_{i=0}^{|\rho|-1} \ln \hat{\tau}_{a_i}(s_i)(s_{i+1}) \cdot l(\mathcal{M}_m; \rho) \end{aligned}$$

Now we optimise with Lagrange multipliers (l_π and $l_{\tau_s^a}$). Let $L(\mathcal{M}, \mathcal{M}_m)$ be the Lagrangian:

$$L(\mathcal{M}, \mathcal{M}_m) = Q(\mathcal{M} \mid \mathcal{M}_m) - l_\pi \left(\sum_{s \in S} \hat{\pi}_s - 1 \right) - \sum_{s \in S} \sum_{a \in A} l_{\tau_s^a} \left(\sum_{u \in S} \hat{\tau}_a(s)(u) - 1 \right)$$

Estimation of π This step is identical to the one for MC. Therefore, we have:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{s' \in S} \gamma_o(s', 0)}$$

Estimation of τ Now, we focus on the transition probability distributions $\tau_a(s)$:

$$\begin{aligned} \frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial \hat{\tau}_{s,a,s'}} &= \frac{\partial}{\partial \hat{\tau}_{s,a,s'}} \left(\sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \sum_{i=0}^{|\rho|-1} \ln [\hat{\tau}_{a_i}(s_i)(s_{i+1})] l(\mathcal{M}_m; \rho) \right) - l_{\tau_s^a} = 0 \\ &= \frac{\partial}{\partial \hat{\tau}_{s,a,s'}} \left(\sum_{\substack{o \in \mathcal{O} \\ u, u' \in S \\ a' \in A}} \sum_{i=0}^{|\rho|-1} \ln [\hat{\tau}_{u,a',u'}] Pr^{\mathcal{M}_m}(X_i = u, X_{i+1} = u', A_i = a', O_{|o|} = o) \right) - l_{\tau_s^a} = 0 \\ &= \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} \frac{1_a(a_i) Pr^{\mathcal{M}_m}(X_i = s, X_{i+1} = s', O_{|o|} = o)}{\hat{\tau}_a(s)(s')} - l_{\tau_s^a} = 0 \end{aligned}$$

Hence:

$$\hat{\tau}_a(s)(s') = \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\rho|-1} \frac{1_a(a_i) Pr^{\mathcal{M}_m}(X_i = s, X_{i+1} = s', O_{|o|} = o)}{l_{\tau_s^a}} \quad (\text{A.12})$$

Furthermore:

$$\frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial l_{\tau_s^a}} = - \left(\sum_{u \in S} \hat{\tau}_a(s)(u) - 1 \right) = 0 \quad (\text{A.13})$$

By plugging (A.12) into (A.13) we get:

$$l_{\tau_s^a} = \sum_{o \in \mathcal{O}} \sum_{u \in S} \sum_{i=0}^{|o|-1} Pr^{\mathcal{M}_m}(X_i = s, X_{i+1} = u, O_{|o|} = o) \cdot 1_a(a_i) \quad (\text{A.14})$$

$$= \sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} Pr^{\mathcal{M}_m}(X_i = s, O_{|o|} = o) \cdot 1_a(a_i) \quad (\text{A.15})$$

And by plugging (A.15) into (A.12):

$$\begin{aligned} \hat{\tau}_a(s)(s') &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} 1_a(a_i) \cdot Pr^{\mathcal{M}_m}(X_i = s, X_{i+1} = s', O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} 1_a(a_i) \cdot Pr^{\mathcal{M}_m}(X_i = s, O_{|o|} = o)} \\ \hat{\tau}_a(s)(s') &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} 1_a(a_i) \cdot Pr^{\mathcal{M}_m}(X_i = s, X_{i+1} = s' \mid O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} 1_a(a_i) \cdot Pr^{\mathcal{M}_m}(X_i = s \mid O_{|o|} = o)} \end{aligned}$$

Finally, using the previously defined coefficients:

$$\hat{\tau}_a(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \xi_o(s, i)(s') \cdot 1_a(a_i)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i) \cdot 1_a(a_i)}$$

Conclusion For MDPs, the UPDATE procedure updates π and τ as follows:

$$\begin{aligned} \hat{\pi}(s) &= \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{u \in S} \gamma_o(u, 0)} \\ \hat{\tau}_a(s)(s') &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \xi_o(s, i)(s') \cdot 1_a(a_i)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|o|-1} \gamma_o(s, i) \cdot 1_a(a_i)} \end{aligned}$$

Remark A.3.1. As for MCs, one may incur in the situation where $\sum_{o \in \mathcal{O}} \sum_{i=1}^{|o|-1} \gamma_o(s, i) \cdot 1_a(a_i) = 0$, indicating that the state s does not play a role in the observed dynamics. In this case the update procedure leaves the distribution $\tau_a(s)$ unchanged.

A.4 Baum-Welch for CTMCs

The forward-backward functions For $o = (\ell_0, t_0) \dots \ell_T$ a timed trace and a CTMC \mathcal{M} , we define the forward and the backward functions $\alpha_o, \beta_o : S \times \{0 \dots T\} \rightarrow [0, 1]$ as

$$\begin{aligned} \alpha_o(s, i) &= Pr^{\mathcal{M}}[Y_{0:i} = \ell_0 \dots \ell_i, X_i = s \mid T_{0:i-1} = t_0 \dots t_{i-1}], \text{ and} \\ \beta_o(s, i) &= Pr^{\mathcal{M}}[Y_{i:T} = \ell_i \dots \ell_T \mid T_{i:T-1} = t_i \dots t_{T-1}, X_i = s]. \end{aligned}$$

These can be calculated according to the following recurrences

$$\alpha_o(s, i) = \begin{cases} 1_{\ell_0}(\ell(s)) \cdot \pi(s) & \text{if } i = 0 \\ 1_{\ell_i}(\ell(s)) \cdot \sum_{s' \in S} \alpha_o(s', i-1) \cdot \tau(s')(s) \cdot \lambda_{s'} e^{-\lambda_{s'} t_i} & \text{if } 0 < i \leq T \end{cases}$$

$$\beta_o(s, i) = \begin{cases} 1_{\ell_T}(\ell(s)) & \text{if } i = T \\ 1_{\ell_i}(\ell(s)) \cdot \sum_{s' \in S} \tau(s)(s') \cdot \lambda_s e^{-\lambda_s t_i} \cdot \beta_o(s', i+1) & \text{if } 0 \leq i < T \end{cases}$$

Thus:

$$\gamma_o(s, i) = \frac{\alpha_o(s, i) \beta_o(s, i)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

$$\xi_o(s, i)(s') = \frac{\alpha_o(s, i) \cdot \lambda_s e^{-\lambda_s t_i} \cdot \tau(s)(s') \cdot \beta_o(s', i+1)}{\sum_{u \in S} \alpha_o(u, i) \beta_o(u, i)}$$

Additionally, we have:

$$\ln l(\mathcal{M}; \rho) = \underbrace{\ln \pi(s_0)}_{\text{initial state}} + \underbrace{\sum_{i=0}^{|\rho|-1} \ln(\lambda_{s_i} e^{-\lambda_{s_i} t_i})}_{\text{dwell times}} + \underbrace{\sum_{i=0}^{|\rho|-1} \ln \tau(s_i)(s_{i+1})}_{\text{transitions}} \quad (\text{A.16})$$

The Lagrangian multipliers On a given finite set \mathcal{O} of traces, the Baum-Welch algorithm can be described as repeating the two following steps until convergence:

1. (E step) Compute $Q(\mathcal{M} | \mathcal{M}_m) = \sum_{o \in \mathcal{O}} \sum_{\rho \in \text{Paths}(o)} \ln [l(\mathcal{M}; \rho)] l(\mathcal{M}_m; \rho)$.
2. (M step) Set $\mathcal{M}_{m+1} = \arg \max_{\mathcal{M}} Q(\mathcal{M} | \mathcal{M}_m)$.

Let $\mathcal{M}_m = \langle S, \mathcal{L}, \ell, R, \pi \rangle$ and $\mathcal{M} = \langle S, \mathcal{L}, \ell, \hat{R}, \hat{\pi} \rangle$. Recall that $E(s) = \sum_{s' \in S} R(s)(s')$ and $\hat{E}(s) = \sum_{s' \in S} \hat{R}(s)(s')$.

We define $\tau(s)(s') = R(s)(s')/E(s)$, $\hat{\tau}(s)(s') = \hat{R}(s)(s')/\hat{E}(s)$, $\lambda_s = E(s)$ and $\hat{\lambda}_s = \hat{E}(s)$ as above.

First, plugging (A.16) into $Q(\mathcal{M} | \mathcal{M}_m)$, we get:

$$\begin{aligned} Q(\mathcal{M} | \mathcal{M}_m) &= \sum_{o \in \mathcal{O}} \sum_{\rho \in \text{Paths}(o)} \ln \hat{\pi}(s_0) \cdot l(\mathcal{M}_m; \rho) \\ &+ \sum_{o \in \mathcal{O}} \sum_{\rho \in \text{Paths}(o)} \sum_{i=0}^{|\rho|-1} \ln \hat{\tau}(s_i)(s_{i+1}) \cdot l(\mathcal{M}_m; \rho) \\ &+ \sum_{o \in \mathcal{O}} \sum_{\rho \in \text{Paths}(o)} \sum_{i=0}^{|\rho|-1} \ln(\lambda_{s_i} e^{-\lambda_{s_i} t_i}) \cdot l(\mathcal{M}_m; \rho) \end{aligned}$$

Now we optimise with Lagrange multipliers (l_π and l_{τ_s}). Let $L(\mathcal{M}, \mathcal{M}_m)$ be the Lagrangian:

$$L(\mathcal{M}, \mathcal{M}_m) = Q(\mathcal{M} | \mathcal{M}_m) - l_\pi \left(\sum_{s \in S} \hat{\pi}(s) - 1 \right) - \sum_{s \in S} l_{\tau_s} \left(\sum_{s' \in S} \hat{\tau}(s)(s') - 1 \right)$$

Estimation of π This step is identical to the one for MC. Therefore, we have:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{s' \in S} \gamma_o(s', 0)}$$

Estimation of τ As for π , this step is identical to the one for MC. Therefore:

$$\hat{\tau}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|-1} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=1}^{|\mathcal{O}|} \gamma_o(s, i)}$$

Estimation of λ First, we notice that:

$$\ln(\lambda_s e^{-\lambda_s t}) = \ln \lambda_s - \lambda_s t$$

We start as usual by deriving the Lagrangian with respect to $\hat{\lambda}_s$:

$$\begin{aligned} \frac{\partial L(\mathcal{M}, \mathcal{M}_m)}{\partial \hat{\lambda}_s} &= \frac{\partial Q(\mathcal{M} | \mathcal{M}_m)}{\partial \hat{\lambda}_s} = 0 \\ &= \frac{\partial}{\partial \hat{\lambda}_s} \left(\sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \sum_{i=0}^{|\rho|} (\ln \lambda_{s_i} - \lambda_{s_i} t_i) \cdot l(\mathcal{M}_m; \rho) \right) = 0 \\ &= \frac{\partial}{\partial \hat{\lambda}_s} \left(\sum_{o \in \mathcal{O}} \sum_{s' \in S} \sum_{i=0}^{|\mathcal{O}|} (\ln \lambda_{s'} - \lambda_{s'} t_i) \cdot Pr^{\mathcal{M}_m}(X_i = s', O_{|o|} = o) \right) = 0 \\ &= \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \left(\frac{1}{\lambda_s} - t_i \right) Pr^{\mathcal{M}_m}(X_i = s, O_{|o|} = o) = 0 \end{aligned}$$

Hence:

$$\begin{aligned} \hat{\lambda}_s &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} Pr^{\mathcal{M}_m}(X_i = s, O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i Pr^{\mathcal{M}_m}(X_i = s, O_{|o|} = o)} \\ \hat{\lambda}_s &= \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} Pr^{\mathcal{M}_m}(X_i = s | O_{|o|} = o)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i Pr^{\mathcal{M}_m}(X_i = s | O_{|o|} = o)} \end{aligned}$$

Finally, using the previously defined coefficients:

$$\hat{\lambda}_s = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \gamma_o(s, i)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i \gamma_o(s, i)}$$

Estimation of R We know that

$$\hat{R}(s)(s') = \hat{\tau}(s)(s') \cdot \hat{E}(s) = \hat{\tau}(s)(s') \cdot \hat{\lambda}(s)$$

Then

$$\hat{R}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=1}^{|\mathcal{O}|} \gamma_o(s, i)} \cdot \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \gamma_o(s, i)}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i \gamma_o(s, i)}$$

Hence

$$\hat{R}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i \gamma_o(s, i)}$$

Conclusion Given a set of timed traces \mathcal{O} , the UPDATE procedure updates π and R as follows:

$$\hat{\pi}(s) = \frac{\sum_{o \in \mathcal{O}} \gamma_o(s, 0)}{\sum_{o \in \mathcal{O}} \sum_{u \in S} \gamma_o(u, 0)}$$

$$\hat{R}(s)(s') = \frac{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} \xi_o(s, i)(s')}{\sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|} t_i \gamma_o(s, i)}$$

Appendix B

Proof of Theorem 6.4.1

Let first recall the context and the Theorem.

$g(\mathbf{x}|\mathbf{x}_m) = \sum_{i=1}^n g(x_i|\mathbf{x}_m)$ where

$$g(x_i|\mathbf{x}_m) = \sum_{\omega \in \mathcal{X}} \xi_\omega a_{\omega i} \ln x_i - \sum_s \sum_{\omega \in \mathcal{S}_s} \frac{f_\omega(\mathbf{x}_m) a_{\omega i} \gamma_s}{a_\omega(x_{mi})^{a_\omega}} x_i^{a_\omega}. \quad (\text{B.1})$$

Here the coefficients γ_s and ξ_ω are respectively defined as

$$\gamma_s = \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|-1} \gamma_o(s, i) (\llbracket t_i \neq \emptyset \rrbracket t_i + \llbracket t_i = \emptyset \rrbracket E_m(s)^{-1}) \quad (\text{B.2})$$

$$\xi_\omega = \sum_{o \in \mathcal{O}} \sum_{i=0}^{|\mathcal{O}|-1} \xi_o(\omega, i) \quad (\text{B.3})$$

Theorem (6.4.1). *The surrogate function $g(\mathbf{x}|\mathbf{x}_m)$ minorises $\ln l(\mathbf{x})$ at \mathbf{x}_m up to an irrelevant constant.*

The art of devising an MM algorithm revolves around intelligent choices of minorising functions, which are used to identify a surrogate function. The construction of the surrogate function $g(\mathbf{x}|\mathbf{x}_m)$ (cf. Equation B.1) relies on three inequalities.

The first basic minorisation builds upon Jensen's inequality. For $x_i > 0$, $y_i > 0$ ($i = 1 \dots n$),

$$\ln \left(\sum_{i=1}^n x_i \right) \geq \sum_{i=1}^n \frac{y_i}{\sum_{j=1}^n y_j} \ln \left(\frac{\sum_{j=1}^n y_j}{y_i} x_i \right) \quad (\text{B.4})$$

Note that the above inequality becomes an equality whenever $x_i = y_i$ for all $i = 1 \dots n$. Remarkably, the EM algorithm [21] is a special case of the MM algorithm which revolves around the above basic minorisation when additionally the values x_i and y_i describe a probability distribution, i.e., $\sum_{i=1}^n x_i = 1$ and $\sum_{i=1}^n y_i = 1$.

Our second basic minorisation derives from the strict concavity of the logarithm function, which implies for $x, y > 0$ that

$$-\ln x \geq 1 - \ln y - x/y \quad (\text{B.5})$$

with equality if and only if $x = y$. Note that the above inequality restates the supporting hyperplane property of the convex function $-\ln x$. Notably, the above minorisation is used in [133] for finding rankings in the Bradley–Terry model.

The third basic minorisation [23] derives from the generalised arithmetic-geometric mean inequality which implies, for positive x_i, y_i , and α_i and $\alpha = \sum_{i=1}^n \alpha_i$, that

$$-\prod_{i=1}^n x_i^{\alpha_i} \geq -\left(\prod_{i=1}^n y_i^{\alpha_i}\right) \sum_{i=1}^n \frac{\alpha_i}{\alpha} \left(\frac{x_i}{y_i}\right)^{\alpha}. \quad (\text{B.6})$$

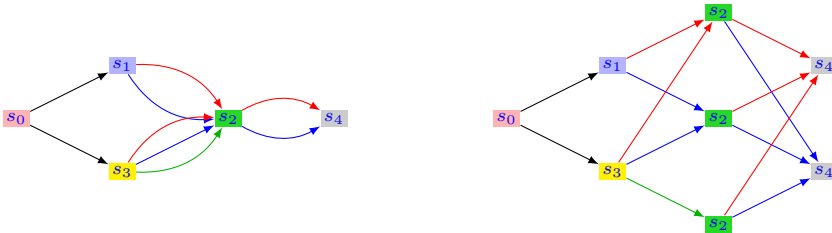
Note again that equality holds when all $x_i = y_i$.

Theorem 6.4.1. For convenience, we establish the result for a pCTMC \mathcal{P} that satisfies the following assumptions:

- (A) there is at most one transition between each pair of states;
- (B) for each transition $\omega \in \rightarrow$, the map f_ω is either a constant (i.e., $f_\omega(\mathbf{x}) = c_\omega$ with $c_\omega \geq 0$) or of the form $f_\omega(\mathbf{x}) = c_\omega \prod_{i=1}^n x_i^{a_{\omega i}}$ where $c_\omega > 0$ and $a_{\omega i} > 0$.

Assumption (B) does not restrict the generality of the formulation. Indeed, a transition $\omega = (s \xrightarrow{f_\omega} s')$ where $f_\omega(\mathbf{x}) = \sum_{j=1}^m c_j \prod_{i=1}^n x_i^{a_{ji}}$ can be replaced by m transitions of the form $\omega_i = (s \xrightarrow{f_{\omega_i}} s')$ where $f_{\omega_i}(\mathbf{x}) = c_j \prod_{i=1}^n x_i^{a_{ji}}$ ($j = 1 \dots m$). Note that in case $c_j = 0$ or $a_{\omega_i} = 0$ the resulting map simplifies to a constant.

As for (A), let us denote by $m(s) = \max_{s' \in S} |\{\omega \mid \omega = (s' \xrightarrow{f} s)\}|$ the maximum number of transitions to s from a single state $s' \in S$. We construct a pCTMC as follows: each state s is duplicated $m(s)$ times, and each state s' having more than one transition to s will redirect each transition to a distinct copy of s . Below is shown an example of such construction. To simplify the drawing we omitted the transition rates. Instead, the transitions that are redirected are highlighted with different colors.



Note that the construction described above ensures that any state s is bisimilar to any of its copies. As a consequence, for any valuation of the parameters, the two chains have the same likelihood value.

Starting from the log-likelihood function, we proceed with the following minori-

sation steps

$$\begin{aligned}
\ln l(\mathbf{x}) &= \sum_{o \in \mathcal{O}} \ln l(\mathcal{P}(\mathbf{x}); o) = \sum_{o \in \mathcal{O}} \ln \left(\sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{P}(\mathbf{x}); \rho) \right) \\
&\geq \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \frac{l(\mathcal{P}(\mathbf{x}_m); \rho)}{l(\mathcal{P}(\mathbf{x}_m); o)} \ln \left(\frac{l(\mathcal{P}(\mathbf{x}_m); o)}{l(\mathcal{P}(\mathbf{x}_m); \rho)} \cdot l(\mathcal{P}(\mathbf{x}); \rho) \right) && \text{(by (B.4))} \\
&\cong \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{P}(\mathbf{x}_m), o; \rho) \ln(l(\mathcal{P}(\mathbf{x}); \rho)) && \text{(up-to const)}
\end{aligned}$$

using Equation (6.3) we simplify the above to

$$\begin{aligned}
&= \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{P}(\mathbf{x}_m), o; \rho) \ln \left(\pi(s_0) \prod_{i=0}^{|\rho|-1} \frac{R(s_i, s_{i+1})}{E(s_i)} \cdot \prod_{i \in \mathcal{T}(o)} E(s_i) e^{-E(s_i)t_i} \right) \\
&= \sum_{o \in \mathcal{O}} \sum_{\rho \in \mathbf{Paths}(o)} \sum_{i=0}^{|\rho|-1} l(\mathcal{P}(\mathbf{x}_m), o; \rho) (\ln \pi(s_0) + \ln R(s_i, s_{i+1}) \\
&\quad - \llbracket t_i = \emptyset \rrbracket \ln E(s_i) - \llbracket t_i \neq \emptyset \rrbracket E(s_i)t_i)
\end{aligned} \tag{B.7}$$

By (B.5) we minorise $-\ln E(s_i)$ up-to some constant as $-E(s_i)/E_m(s_i)$. Let $\delta_s(t) = (\llbracket t_i = \emptyset \rrbracket E_m(s)^{-1} + \llbracket t_i \neq \emptyset \rrbracket t_i)$, then the overall minorisation simplifies to

$$\begin{aligned}
&\geq \sum_{o \in \mathcal{O}} \sum_{t=0}^{|\rho|-1} \sum_{\rho \in \mathbf{Paths}(o)} l(\mathcal{P}(\mathbf{x}_m), o; \rho) (\ln \pi(s_0) + \ln R(s_i, s_{i+1}) - E(s_i)\delta_{s_i}(t)) \\
&= \sum_{o \in \mathcal{O}} \sum_{t=0}^{|\rho|-1} \left(\sum_{\omega \in \rightarrow} \xi_\omega(t) \ln f_\omega(\mathbf{x}) - \sum_s \gamma_s(t) \delta_s(t) E(s) \right) && \text{(def. } \xi_\omega(t) \text{ and } \gamma_s(t)) \\
&\cong \sum_{\omega \in \rightarrow} \xi_\omega \ln f_\omega(\mathbf{x}) + \sum_s \gamma_s(-E(s)) \\
&\hspace{15em} \text{(rearrange up-to const, using (B.2) and (B.3))} \\
&\cong \sum_{i=1}^n \sum_{\omega \in \rightarrow} \xi_\omega a_{\omega i} \ln x_i + \sum_s \gamma_s(-E(s)) && \text{((B), up-to const)} \\
&\geq \sum_{i=1}^n \left[\sum_{\omega \in \rightarrow} \xi_\omega a_{\omega i} \ln x_i - \sum_s \sum_{\omega \in s \rightarrow} \frac{f_\omega(\mathbf{x}_m) a_{\omega i} \gamma_s}{a_\omega x_{mi}^{a_\omega}} x_i^{a_\omega} \right] && (**) \\
&= g(\mathbf{x} | \mathbf{x}_m) && \text{(by (B.1))}
\end{aligned}$$

Where (**) is justified by the following minorisation of $-E(s)$

$$\begin{aligned}
-E(s) &= \sum_{\omega \in s \rightarrow \cdot} -f_{\omega}(\mathbf{x}) \cong \sum_{\omega \in s \xrightarrow{\mathbf{x}} \cdot} c_{\omega} \left(-\prod_{i=1}^n x_i^{a_{\omega i}} \right) \quad (\text{up-to const, by ((B))}) \\
&\geq \sum_{\omega \in s \xrightarrow{\mathbf{x}} \cdot} -c_{\omega} \left(\prod_{i=1}^n x_{mi}^{a_{\omega i}} \right) \sum_{i=1}^n \frac{a_{\omega i}}{a_{\omega}} \left(\frac{x_i}{x_{mi}} \right)^{a_{\omega}} \quad (\text{by (B.6)}) \\
&\geq -\sum_{i=1}^n \sum_{\omega \in s \xrightarrow{\mathbf{x}} \cdot} \frac{f_{\omega}(\mathbf{x}_m) a_{\omega i}}{a_{\omega} x_{mi}^{a_{\omega}}} x_i^{a_{\omega}} \quad (\text{rearranging})
\end{aligned}$$

As shown above, there exists a (non-negative) constant c such that the surrogate function $g(\mathbf{x}|\mathbf{x}_m) + c$ minorises $\ln l(\mathbf{x})$ at \mathbf{x}_m . \square

Appendix C

Scalability evaluation over training set size

We repeat here the experiment presented in Section 7.5 but this time with training set two times bigger.

Utilising JAJAPY, we learn MCs ranging from 10 to 200 states (models with 100 to 40,000 parameters). Training sets comprise 2,000 traces of length 10; the initial hypothesis size matches the SUL. Results are depicted in Fig. C.1.

Comparing with prior Section 7.5, we note slower and more memory-intensive learning from a doubled training set. Intriguingly, doubling trace numbers cuts loglikelihood distance in half. In essence, learning from an x times larger set yields a model x times closer to the SUL (in loglikelihood distance).

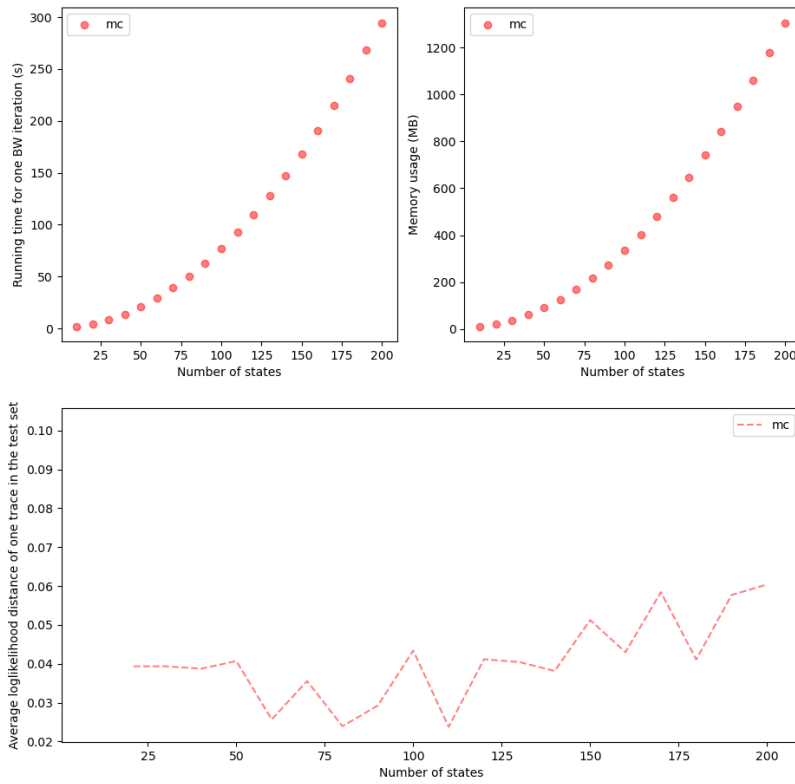


Figure C.1 – JAJAPY running time, memory usage and loglikelihood distance w.r.t. the number of parameters of the hypothesis.

Bibliography

- [1] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruijters, “The quantitative verification benchmark set,” in *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Proceedings, Part I* (T. Vojnar and L. Zhang, eds.), vol. 11427 of *Lecture Notes in Computer Science*, pp. 344–350, Springer, 2019. ix, 67
- [2] A. Vodencarevic, A. Maier, and O. Niggemann, “Evaluating learning algorithms for stochastic finite automata: Comparative empirical analyses on learning models for technical systems,” *ICPRAM 2013 - Proceedings of the 2nd International Conference on Pattern Recognition Applications and Methods*, pp. 229–238, 01 2013. xi, 33
- [3] D. Knuth and A. Yao, *Algorithms and Complexity: New Directions and Recent Results*, ch. The complexity of nonuniform random number generation. Academic Press, 1976. xi, 81
- [4] K. Lueth, “State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time,” 11 2020. [Online; posted 19-November-2020]. 1
- [5] S. Baase, *Principles of model checking*. Pearson Prentice Hall, 2008. 1
- [6] C. Isidore, “Boeing’s 737 max debacle could be the most expensive corporate blunder ever,” November 2020. [Online; posted 17-November-2020]. 1
- [7] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008. 2, 5, 6, 10, 12, 42, 50
- [8] E. A. Emerson and E. M. Clarke, “Characterizing correctness properties of parallel programs using fixpoints,” in *Automata, Languages and Programming, 7th Colloquium, Noordwijkerhout, The Netherlands, July 14-18, 1980, Proceedings* (J. W. de Bakker and J. van Leeuwen, eds.), vol. 85 of *Lecture Notes in Computer Science*, pp. 169–181, Springer, 1980. 5
- [9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986. 5
- [10] D. L. Dill, “A retrospective on murphi,” in *25 Years of Model Checking - History, Achievements, Perspectives* (O. Grumberg and H. Veith, eds.), vol. 5000 of *Lecture Notes in Computer Science*, pp. 77–88, Springer, 2008. 5
- [11] G. J. Holzmann, “The model checker SPIN,” *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997. 5

- [12] M. Z. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of Probabilistic Real-Time Systems,” in *CAV 2011*, vol. 6806 of *LNCS*, pp. 585–591, Springer, 2011. 5, 47, 57, 74
- [13] C. Dehnert, S. Junges, J. Katoen, and M. Volk, “A Storm is Coming: A Modern Probabilistic Model Checker,” in *CAV 2017*, vol. 10427 of *LNCS*, pp. 592–600, Springer, 2017. 5, 57, 74
- [14] E. M. Hahn, A. Hartmanns, C. Hensel, M. Klauck, J. Klein, J. Křetínský, D. Parker, T. Quatmann, E. Ruijters, and M. Steinmetz, “The 2019 comparison of tools for the analysis of quantitative formal models,” in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Beyer, M. Huisman, F. Kordon, and B. Steffen, eds.), (Cham), pp. 69–92, Springer International Publishing, 2019. 5
- [15] P. Billingsley, *Probability and Measure*. Wiley, 1995. 6
- [16] J. Stigler, F. Ziegler, A. Gieseke, J. C. M. Gebhardt, and M. Rief, “The complex folding network of single calmodulin molecules,” *Science*, vol. 334, no. 6055, pp. 512–516, 2011. 16
- [17] K.-C. Wong, T.-M. Chan, C. Peng, Y. Li, and Z. Zhang, “DNA motif elucidation using belief propagation,” *Nucleic Acids Research*, vol. 41, pp. e153–e153, 06 2013. 16
- [18] S. Shah, A. K. Dubey, and J. Reif, “Improved optical multiplexing with temporal dna barcodes,” *ACS Synthetic Biology*, vol. 8, no. 5, pp. 1100–1111, 2019. PMID: 30951289. 16
- [19] A. Petropoulos, S. P. Chatzis, and S. Xanthopoulos, “A novel corporate credit rating system based on student’s-t hidden markov models,” *Expert Systems with Applications*, vol. 53, pp. 87–105, 2016. 16
- [20] C. Karlof and D. Wagner, “Hidden markov model cryptanalysis,” in *Cryptographic Hardware and Embedded Systems - CHES 2003* (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), (Berlin, Heidelberg), pp. 17–34, Springer Berlin Heidelberg, 2003. 16
- [21] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum likelihood from incomplete data via the em algorithm,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977. 25, 27, 59, 105
- [22] K. Lange, *MM Optimization Algorithms*. SIAM, 2016. 25, 59, 64, 65
- [23] K. Lange, *Optimization*. Springer New York, NY, 2 ed., 2013. 25, 59, 64, 65, 106
- [24] G. C. G. Wei and M. A. Tanner, “A monte carlo implementation of the em algorithm and the poor man’s data augmentation algorithms,” *Journal of the American Statistical Association*, vol. 85, no. 411, pp. 699–704, 1990. 25
- [25] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, “A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains,” *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164 – 171, 1970. 25, 77

- [26] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proc. IEEE*, vol. 77, pp. 257–286, 1989. 25, 26, 28, 59, 60, 74, 77, 78
- [27] L. Rabiner, "First-hand:the hidden markov model," *IEEE Global History Network*, 2013. 25, 28
- [28] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 35–49, 2008. 26
- [29] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *Advances in Cryptology – ASIACRYPT 2009* (M. Matsui, ed.), (Berlin, Heidelberg), pp. 667–684, Springer Berlin Heidelberg, 2009. 26
- [30] S. L. Salzberg, A. L. Delcher, S. Kasif, and O. White, "Microbial gene identification using interpolated markov models.," *Nucleic acids research*, vol. 26 2, pp. 544–8, 1998. 26
- [31] D. Hsu, S. M. Kakade, and T. Zhang, "A spectral algorithm for learning hidden markov models," 2012. 26
- [32] B. Balle, A. Quattoni, and X. Carreras, "Local loss optimization in operator models: A new insight into spectral learning," 2012. 26
- [33] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. 26, 37, 42, 50, 75, 78, 85
- [34] R. Neal, "Markov chain sampling methods for dirichlet process mixture models," *Journal of Computational and Graphical Statistics*, vol. 9, 01 2000. 26, 75
- [35] M. Araya-López, O. Buffet, V. Thomas, and F. Charpillet, "Active Learning of MDP Models," in *European Workshop On Reinforcement Learning*, (Athènes, Greece), Sept. 2011. 26
- [36] B. R. Ceppellini, M. Siniscalco, and C. A. B. Smith, "The estimation of gene frequencies in a random-mating population," *Annals of Human Genetics*, vol. 20, no. 2, pp. 97–115, 1955. 27
- [37] H. O. Hartley, "Maximum likelihood estimation from incomplete data," *Biometrics*, vol. 14, no. 2, pp. 174–194, 1958. 27
- [38] C. F. J. Wu, "On the convergence properties of the em algorithm," *The Annals of Statistics*, vol. 11, no. 1, pp. 95–103, 1983. 27, 28, 93
- [39] R. M. Neal and G. E. Hinton, *A View of the Em Algorithm that Justifies Incremental, Sparse, and other Variants*, pp. 355–368. Dordrecht: Springer Netherlands, 1998. 28
- [40] F. Yang, S. Balakrishnan, and M. J. Wainwright, "Statistical and computational guarantees for the baum-welch algorithm," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 658–665, 2015. 28

- [41] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "On the memory complexity of the forward-backward algorithm," *Pattern Recognition Letters*, vol. 31, no. 2, pp. 91–99, 2010. 28, 87
- [42] J. Raviv, "Decision making in markov chains applied to the problem of pattern recognition," *IEEE Transactions on Information Theory*, vol. IT-3, no. 4, p. 536 – 551, 1967. 28
- [43] G. Ott, "Compact encoding of stationary markov sources," *IEEE Transactions on Information Theory*, vol. 13, no. 1, p. 82 – 86, 1967. 28
- [44] B. Vanluyten, J. C. Willems, and B. D. Moor, "A new approach for the identification of hidden markov models," *2007 46th IEEE Conference on Decision and Control*, pp. 4901–4905, 2007. 28
- [45] T. Liu and J. Lemeire, "Efficient and effective learning of hmms based on identification of hidden states," *Mathematical Problems in Engineering*, vol. 2017, Feb. 2017. 32
- [46] R. C. Carrasco and J. Oncina, "Learning Stochastic Regular Grammars by Means of a State Merging Method," in *ICGI-94* (R. C. Carrasco and J. Oncina, eds.), vol. 862 of *LNCS*, pp. 139–152, Springer, 1994. 34, 35, 36, 37, 42, 74, 75, 78
- [47] R. C. Carrasco and J. Oncina, "Learning deterministic regular grammars from stochastic samples in polynomial time," *RAIRO – Theoretical Informatics and Applications (RAIRO: ITA)*, vol. 33, no. 1, pp. 1–20, 1999. 34, 42, 74, 75, 78
- [48] K. Sen, M. Viswanathan, and G. Agha, "Learning continuous time markov chains from sample executions," in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pp. 146–155, 2004. 34, 35, 36, 37, 59
- [49] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, "Learning Deterministic Probabilistic Automata from a Model Checking Perspective," *Machine Learning*, vol. 105, no. 2, pp. 255–299, 2012. 34, 36, 37, 42, 74, 78
- [50] Y. Chen and T. D. Nielsen, "Active learning of markov decision processes for system verification," *2012 11th International Conference on Machine Learning and Applications*, vol. 2, pp. 289–294, 2012. 34
- [51] F. Thollard, P. Dupont, and C. de la Higuera, "Probabilistic dfa inference using kullback-leibler divergence and minimality," in *International Conference on Machine Learning*, 2000. 34, 37, 75
- [52] W. Hoeffding, *Probability Inequalities for sums of Bounded Random Variables*, pp. 409–426. New York, NY: Springer New York, 1994. 34, 78
- [53] D. R. Cox, "Some simple approximate tests for poisson," *Biometrika*, vol. 40, pp. 354–360, 12 1953. 36
- [54] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951. 37

- [55] T. Margaria, O. Niese, H. Raffelt, and B. Steffen, “Efficient test-based model generation for legacy reactive systems,” in *Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International, HLDVT '04*, (USA), p. 95–100, IEEE Computer Society, 2004. 38
- [56] M. Shahbaz and R. Groz, “Inferring Mealy Machines,” in *Formal Methods 2009*, (Eindhoven, Netherland), pp. 207–222, 2009. 38
- [57] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, “Active learning for extended finite state machines,” *Formal Aspects Comput.*, vol. 28, no. 2, pp. 233–263, 2016. 38, 50
- [58] A. Khalili and A. Tacchella, “Learning nondeterministic mealy machines,” in *International Conference on Graphics and Interaction*, 2014. 38
- [59] M. Tappler, B. K. Aichernig, G. Bacci, M. Eichlseder, and K. G. Larsen, “L^{*}-based learning of markov decision processes (extended version),” *Formal Aspects Comput.*, vol. 33, no. 4-5, pp. 575–615, 2021. 38, 39, 50, 52, 54
- [60] B. Steffen, F. Howar, and M. Merten, “Introduction to active automata learning from a practical perspective,” in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, 2011. 39, 50
- [61] E. Muškardin, B. Aichernig, I. Pill, A. Pferscher, and M. Tappler, “Aalpy: an active automata learning library,” *Innovations in Systems and Software Engineering*, vol. 18, pp. 1–10, 03 2022. 41, 42, 54, 75
- [62] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition, Addison-Wesley, 2007. 42
- [63] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, 1994. 42
- [64] J. Oncina and P. García, “Inferring regular languages in polynomial update time,” *World Scientific*, 01 1992. 42
- [65] K. El-Fakih, R. Groz, M. N. Irfan, and M. Shahbaz, “Learning Finite State Models of Observable Nondeterministic Systems in a Testing Context,” in *22nd IFIP International Conference on Testing Software and Systems*, (Natal, Brazil), pp. 97–102, 2010. 42
- [66] A. Pferscher and B. K. Aichernig, “Learning abstracted non-deterministic finite state machines,” in *Testing Software and Systems* (V. Casola, A. De Benedictis, and M. Rak, eds.), (Cham), pp. 52–69, Springer International Publishing, 2020. 42
- [67] M. Tappler, E. Muskardin, B. K. Aichernig, and I. Pill, “Active model learning of stochastic reactive systems,” in *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings* (R. Calinescu and C. S. Pasareanu, eds.), vol. 13085 of *Lecture Notes in Computer Science*, pp. 481–500, Springer, 2021. 42, 75, 78, 85

- [68] H. Mao, Y. Chen, M. Jaeger, T. Nielsen, K. Larsen, and B. Nielsen, “Learning probabilistic automata for model checking,” in *Proceedings of the 2011 8th International Conference on Quantitative Evaluation of Systems, QEST 2011*, pp. 111 – 120, 10 2011. 42, 78
- [69] A. Pferscher and B. K. Aichernig, “Fingerprinting and analysis of bluetooth devices with automata learning,” *CoRR*, vol. abs/2211.16074, 2022. 42
- [70] A. Schliep, W. Rungarityotin, B. Georgi, and A. Sch, “The general hidden markov model library: Analyzing systems with unobservable states,” *Proceedings of the Heinz-Billing-Price*, 01 2004. 42
- [71] G. Bacci, A. Ingólfssdóttir, K. G. Larsen, and R. Reynouard, “Active learning of markov decision processes using baum-welch algorithm,” in *20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021, Pasadena, CA, USA, December 13-16, 2021* (M. A. Wani, I. K. Sethi, W. Shi, G. Qu, D. S. Raicu, and R. Jin, eds.), pp. 1203–1208, IEEE, 2021. 47, 48, 71, 74, 77, 78, 86
- [72] G. Bacci, A. Ingólfssdóttir, K. G. Larsen, and R. Reynouard, “An mm algorithm to estimate parameters in continuous-time markov chains,” in *Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 18-23, 2023, Proceedings* (N. J. M. Tribastone, ed.), Lecture Notes in Computer Science, Springer, 2023. 47, 48, 74, 77, 78
- [73] R. Reynouard, A. Ingólfssdóttir, and G. Bacci, “Jajapy: a learning library for stochastic models,” in *Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 18-23, 2023, Proceedings* (N. J. M. Tribastone, ed.), Lecture Notes in Computer Science, Springer, 2023. 48, 67, 74
- [74] M. Isberner, F. Howar, and B. Steffen, “The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning,” in *RV 2014* (B. Bonakdarpour and S. A. Smolka, eds.), vol. 8734 of *LNCS*, pp. 307–322, Springer, 2014. 50
- [75] G. Shani, R. I. Brafman, and S. E. Shimony, “Model-Based Online Learning of POMDPs,” in *ECML*, vol. 3720 of *Lecture Notes in Computer Science*, pp. 353–364, Springer, 2005. 50
- [76] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall, 1984. 57
- [77] J. Hillston, *A compositional approach to performance modelling*. PhD thesis, University of Edinburgh, UK, 1994. 57
- [78] F. Ciocchetta and J. Hillston, “Bio-pepa: A framework for the modelling and analysis of biological systems,” *Theor. Comput. Sci.*, vol. 410, no. 33-34, pp. 3065–3084, 2009. 57
- [79] M. Z. Kwiatkowska, G. Norman, and D. Parker, “Using probabilistic model checking in systems biology,” *SIGMETRICS Perform. Evaluation Rev.*, vol. 35, no. 4, pp. 14–21, 2008. 57
- [80] R. Alur and T. A. Henzinger, “Reactive modules,” *Formal Methods Syst. Des.*, vol. 15, no. 1, pp. 7–48, 1999. 57, 74

- [81] P. Milazzo, “Analysis of covid-19 data with prism: Parameter estimation and sir modelling,” in *From Data to Models and Back* (J. Bowles, G. Broccia, and M. Nanni, eds.), (Cham), pp. 123–133, Springer International Publishing, 2021. 58, 59, 69, 70
- [82] R. Calinescu, M. Ceska, S. Gerasimou, M. Kwiatkowska, and N. Paoletti, “Efficient synthesis of robust models for stochastic systems,” *J. Syst. Softw.*, vol. 143, pp. 140–158, 2018. 59, 62
- [83] T. Han, J. Katoen, and A. Mereacre, “Approximate parameter synthesis for probabilistic time-bounded reachability,” in *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pp. 173–182, IEEE Computer Society, 2008. 59, 60, 62
- [84] S. Terwijn, “On the Learnability of Hidden Markov Models,” in *Grammatical Inference: Algorithms and Applications, 6th International Colloquium: ICGI 2002, Proceedings* (P. W. Adriaans, H. Fernau, and M. van Zaanen, eds.), vol. 2484 of *Lecture Notes in Computer Science*, pp. 261–268, Springer, 2002. 59
- [85] M. Jamshidian and R. I. Jennrich, “Acceleration of the EM Algorithm by Using Quasi-Newton Methods,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 59, no. 3, pp. 569–587, 1997. 59
- [86] H. Zhou, D. H. Alexander, and K. Lange, “A quasi-Newton acceleration for high-dimensional optimization algorithms,” *Stat. Comput.*, vol. 21, no. 2, pp. 261–273, 2011. 59
- [87] N. Geisweiller, *Finding the Most Likely Values inside a PEPA Model According to Partially Observable Executions*. PhD thesis, LAAS, 2006. 59, 60
- [88] A. Georgoulas, J. Hillston, and G. Sanguinetti, “Proppa: Probabilistic programming for stochastic dynamical systems,” *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 1, pp. 3:1–3:23, 2018. 59
- [89] T. S. Badings, N. Jansen, S. Junges, M. Stoelinga, and M. Volk, “Sampling-Based Verification of CTMCs with Uncertain Rates,” in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II* (S. Shoham and Y. Vizel, eds.), vol. 13372 of *Lecture Notes in Computer Science*, pp. 26–47, Springer, 2022. 59, 60
- [90] A. Georgoulas, J. Hillston, D. Milios, and G. Sanguinetti, “Probabilistic programming process algebra,” in *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings* (G. Norman and W. H. Sanders, eds.), vol. 8657 of *Lecture Notes in Computer Science*, pp. 249–264, Springer, 2014. 59
- [91] W. Wei, B. Wang, and D. F. Towsley, “Continuous-time hidden Markov models for network performance evaluation,” *Perform. Evaluation*, vol. 49, no. 1/4, pp. 129–146, 2002. 60
- [92] D. Schnoerr, G. Sanguinetti, and R. Grima, “Approximation and inference methods for stochastic biochemical kinetics—a tutorial review,” *Journal of Physics A: Mathematical and Theoretical*, vol. 50, p. 093001, jan 2017. 60

- [93] P. Loskot, K. Atitey, and L. Mihaylova, “Comprehensive Review of Models and Methods for Inferences in Bio-Chemical Reaction Networks,” *Frontiers in Genetics*, vol. 10, 2019. 60
- [94] A. Andreychenko, L. Mikeev, D. Spieler, and V. Wolf, “Parameter identification for markov models of biochemical reactions,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 83–98, Springer, 2011. 60
- [95] C. Bayer, A. Moraes, R. Tempone, and P. Vilanova, “An efficient forward-reverse expectation-maximization algorithm for statistical inference in stochastic reaction networks,” *Stochastic Analysis and Applications*, vol. 34, no. 2, pp. 193–231, 2016. 60
- [96] R. A. Levine and G. Casella, “Implementations of the monte carlo em algorithm,” *Journal of Computational and Graphical Statistics*, vol. 10, no. 3, pp. 422–439, 2001. 60, 71
- [97] C. Bayer and J. Schoenmakers, “Simulation of forward-reverse stochastic representations for conditional diffusions,” *The Annals of Applied Probability*, vol. 24, no. 5, pp. 1994 – 2032, 2014. 60
- [98] B. J. Daigle, M. K. Roh, L. R. Petzold, and J. Niemi, “Accelerated maximum likelihood parameter estimation for stochastic biochemical systems,” *BMC Bioinform.*, vol. 13, p. 68, 2012. 60
- [99] N. Jansen, S. Junges, and J. Katoen, “Parameter synthesis in markov models: A gentle survey,” in *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday* (J. Raskin, K. Chatterjee, L. Doyen, and R. Majumdar, eds.), vol. 13660 of *Lecture Notes in Computer Science*, pp. 407–437, Springer, 2022. 60
- [100] M. Ceska, F. Dannenberg, N. Paoletti, M. Kwiatkowska, and L. Brim, “Precise parameter synthesis for stochastic biochemical systems,” *Acta Informatica*, vol. 54, no. 6, pp. 589–623, 2017. 60
- [101] H. Hermanns, J. Meyer-Kayser, and M. Siegle, “Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains,” in *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC’99)* (B. Plateau, W. Stewart, and M. Silva, eds.), pp. 188–207, Prentice-Hall, 1999. 63, 68
- [102] L. Ondel, L.-M. Lam-Yee-Mui, M. Kocour, C. F. Corro, and L. Burget, “Gpu-accelerated forward-backward algorithm with application to lattice-free mmi,” in *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8417–8421, 2022. 71, 88
- [103] C. Eisentraut, H. Hermanns, and L. Zhang, “Concurrency and composition in a stochastic world,” in *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings* (P. Gastin and F. Laroussinie, eds.), vol. 6269 of *Lecture Notes in Computer Science*, pp. 21–39, Springer, 2010. 71

- [104] C. Eisentraut, H. Hermanns, and L. Zhang, “On probabilistic automata in continuous time,” in *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pp. 342–351, IEEE Computer Society, 2010. 71
- [105] G. Bacci, G. Bacci, K. G. Larsen, and R. Mardare, “On the Metric-Based Approximate Minimization of Markov Chains,” in *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland* (I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, eds.), vol. 80 of *LIPICs*, pp. 104:1–104:14, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. 71
- [106] B. Balle, C. Lacroce, P. Panangaden, D. Precup, and G. Rabusseau, “Optimal spectral-norm approximate minimization of weighted finite automata,” in *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)* (N. Bansal, E. Merelli, and J. Worrell, eds.), vol. 198 of *LIPICs*, pp. 118:1–118:20, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 71
- [107] R. Reynouard, “Jajapy’s documentation.” <https://jajapy.readthedocs.io/en/latest/>, 2022. 73
- [108] R. Reynouard, “A short introduction to jajapy.” <https://doi.org/10.5281/zenodo.7695105>, Mar. 2023. 73
- [109] C. E. Budde, A. Hartmanns, M. Klauck, J. Křetínský, D. Parker, T. Quatmann, A. Turrini, and Z. Zhang, “On correctness, precision, and performance in quantitative verification - qcomp 2020 competition report,” in *Leveraging Applications of Formal Methods*, 2020. 74
- [110] “Jajapy github repository.” <https://github.com/Rapfff/jajapy>, 2022. 74
- [111] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, “Learning Probabilistic Automata for Model Checking,” in *QEST 2011*, pp. 111–120, IEEE Computer Society, 2011. 74
- [112] I. Sassi, S. Anter, and A. Bekkhoucha, “A new improved baum-welch algorithm for unsupervised learning for continuous-time hmm using spark,” *International Journal of Intelligent Engineering and Systems*, vol. 13, pp. 214–226, 02 2020. 74, 77, 78
- [113] M. Isberner, F. Howar, and B. Steffen, “The open-source learnlib,” in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), (Cham), pp. 487–495, Springer International Publishing, 2015. 75
- [114] B. Bollig, J. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, “libalf: The automata learning framework,” in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings* (T. Touili, B. Cook, and P. B. Jackson, eds.), vol. 6174 of *Lecture Notes in Computer Science*, pp. 360–364, Springer, 2010. 75
- [115] A. Stolcke, “Bayesian learning of probabilistic language models,” 1994. 75

- [116] A. Gelfand and A. Smith, “Sampling-based approaches to calculate marginal densities,” *Journal of the American Statistical Association*, vol. 85, pp. 398–409, 06 1990. 75
- [117] P. Rashidinejad, B. Zhu, C. Ma, J. Jiao, and S. J. Russell, “Bridging offline reinforcement learning and imitation learning: A tale of pessimism,” *IEEE Transactions on Information Theory*, vol. 68, pp. 8156–8196, 2021. 75
- [118] Y. Jin, Z. Yang, and Z. Wang, “Is pessimism provably efficient for offline rl?,” in *International Conference on Machine Learning*, 2020. 75
- [119] J. Buckman, C. Gelada, and M. G. Bellemare, “The importance of pessimism in fixed-dataset policy optimization,” *ArXiv*, vol. abs/2009.06799, 2020. 75
- [120] J. Morimoto and K. Doya, “Robust Reinforcement Learning,” *Neural Computation*, vol. 17, pp. 335–359, 02 2005. 75
- [121] S. H. Lim, H. Xu, and S. Mannor, “Reinforcement learning in robust markov decision processes,” in *Advances in Neural Information Processing Systems* (C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, eds.), vol. 26, Curran Associates, Inc., 2013. 75
- [122] M. Suilen, T. D. Simão, D. Parker, and N. Jansen, “Robust anytime learning of markov decision processes,” 2023. 75
- [123] E. Derman, D. J. Mankowitz, T. A. Mann, and S. Mannor, “A bayesian approach to robust reinforcement learning,” in *Conference on Uncertainty in Artificial Intelligence*, 2019. 75
- [124] M. Češka, C. Hensel, S. Junges, and J.-P. Katoen, “Counterexample-guided inductive synthesis for probabilistic systems,” *Form. Asp. Comput.*, vol. 33, p. 637–667, aug 2021. 75
- [125] T. Quatmann, C. Dehnert, N. Jansen, S. Junges, and J. Katoen, “Parameter synthesis for markov models: Faster than ever,” *CoRR*, vol. abs/1602.05113, 2016. 75
- [126] N. Jansen, S. Junges, and J. Katoen, “Parameter synthesis in markov models: A gentle survey,” in *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday* (J. Raskin, K. Chatterjee, L. Doyen, and R. Majumdar, eds.), vol. 13660 of *Lecture Notes in Computer Science*, pp. 407–437, Springer, 2022. 75
- [127] E. Polgreen, V. B. Wijesuriya, S. Haesaert, and A. Abate, “Data-efficient bayesian verification of parametric markov chains,” in *Quantitative Evaluation of Systems - 13th International Conference, QEST 2016, Quebec City, QC, Canada, August 23-25, 2016, Proceedings* (G. Agha and B. V. Houdt, eds.), vol. 9826 of *Lecture Notes in Computer Science*, pp. 35–51, Springer, 2016. 75
- [128] F. Yang, S. Balakrishnan, and M. J. Wainwright, “Statistical and computational guarantees for the baum-welch algorithm,” *Journal of Machine Learning Research*, vol. 18, no. 125, pp. 1–53, 2017. 77
- [129] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern,

- M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, pp. 357–362, Sept. 2020. 79
- [130] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, “SymPy: symbolic computing in python,” *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. 79
- [131] F. Bartolucci and S. Pandolfi, “Comment on the paper “on the memory complexity of the forward–backward algorithm,” by khreich w., granger e., miri a., sabourin, r.,” *Pattern Recognition Letters*, vol. 38, pp. 15–19, 2014. 88
- [132] Y. Shao, Y. Wang, D. Povey, and S. Khudanpur, “Pychain: A fully parallelized pytorch implementation of LF-MMI for end-to-end ASR,” in *Inter-speech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020* (H. Meng, B. Xu, and T. F. Zheng, eds.), pp. 561–565, ISCA, 2020. 88
- [133] K. Lange, D. R. Hunter, and I. Yang, “Optimization Transfer Using Surrogate Objective Functions,” *Journal of Computational and Graphical Statistics*, vol. 9, no. 1, pp. 1–20, 2000. 106