Jajapy: A Learning Library for Stochastic Models

Raphaël Reynouard[1](✉), Anna Ingólfsdóttir[1], and Giovanni Bacci[2]

[1] Reykjavík University, Reykjavik, Iceland
raphal20@ru.is
[2] Aalborg University, Aalborg, Denmark

Abstract. 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, as well as Markov decision processes.

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.

The paper 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.

Keywords: Machine Learning · Expectation-Maximization · Model Checking · Markov models · Python

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 [1] and Storm [2] 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

Anna Ingólfsdóttir and Raphaël Reynouard are supported by the *Learning and Applying Probabilistic Systems* project (Project nr. 206574-051) funded by the Icelandic Research Fund.

state-based language based on [3]. 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 [4]. 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 paper, we present JAJAPY [5], 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 [6,7] and IOALERGIA [8,9], 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 [10] to learn MCs, MDPs [11,12], and CTMCs [13] 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 [11,12] when the user has the possibility to interact with the system;
(iv) MM algorithms [13] for estimating parameter value in parametric CTMCs (pCTMCs) from a set of (possibly non-timed) traces.

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.

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 [14] that is complemented with a short video-introduction available on Zenodo [15].

Related Work. AALpy [16] is a recent Python library that can learn both non-stochastic and stochastic models. In particular, AALpy can learn MDPs using L^*_{MDP} [17], an extension of Angluin's L^* algorithm [18], and MCs using Alergia [6,7]. In Sect. 5.3, we compare Jajapy and AALpy performance.

Other automata learning frameworks have been developed as well. For example, Learnlib [19] and libalf [20], which learn non-stochastic models. In contrast with these tools, as of now, Jajapy primary focus is on learning Markov models.

In this paper, we present the different learning methods implemented in Jajapy and AALpy. However, other methods also exist, such as the MDI algorithm [21,22], two state-merging based approaches, or the Bayesian method using Gibbs sampling [23] proposed by Neal in [24].

MDPs are extensively used in *reinforcement learning* as in [25–27], and in *robust reinforcement learning* [28] as in [29–31]. 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) [32] 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 [33,34], where the objective is to find some (or all) instances of a given parametric Markov model satisfying a logic formula. In [35], 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.

Paper Outline. We start with a quick introduction to Jajapy's functionalities in Sect. 2, then we explain Jajapy's features from a theoretical perspective in Sect. 3. In Sect. 4, we present some technical aspects of Jajapy, and in Sect. 5 we evaluate our tool and compare it to AALpy.

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 (Fig. 1).

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 are learned from sequences of labels (i.e., sequences of atomic propositions), (ii) CTMCs and pCTMCs are learned from times series of labels, and (iii) MDPs are learned from alternating sequences of actions and labels.

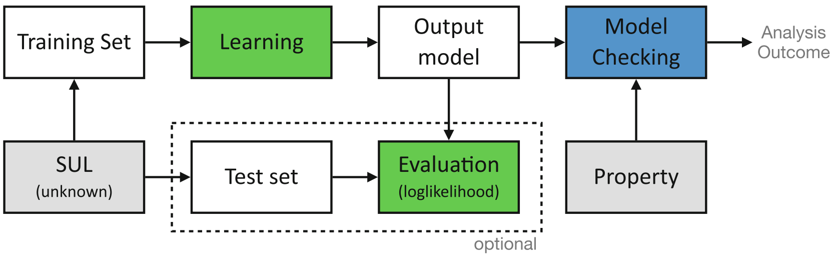

Fig. 1. A complete modeling and verification workflow using Jajapy . The phases where Jajapy is employed are highlighted in green, whereas the phase in blue is assumed to be performed with Storm or Prism (Color figure online).

A trace denotes, depending on the context, a sequence of labels, a time series of labels, and an alternating sequence of actions and labels. The length of a trace is always the number of labels 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 Stormpy sparse model, a model saved in a Prism file or a Jajapy model. Given a training set and an initial hypothesis, the BW algorithm constructs an approximate representation of the SUL, called *output model* (Fig. 2).

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

Fig. 2. Simple execution of Jajapy BW to learn an MC with 10 states.

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.

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.

A Markov model is *deterministic* if, for any state s and label ℓ, there exists at most one transition leaving s to a state labelled with ℓ. Some of the learning methods described below assume the SUL to be deterministic. 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 [36]. This technique is an application of the Expectation Maximisation algorithm. Originally designed for Hidden Markov Models [10], it has been adapted to MCs, CTMCs and MDPs [12,13].

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 [37].

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 [11,12]. 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 [11,12].

State-merging Approach. Both Jajapy and AALpy provide an implementation of the Alergia algorithm [6,7] to learn MCs and its extension IOAlergia [9] 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 [38]. 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} [17], an extension of Angluin's L^* algorithm [18] 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 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.

Table 1. Key characteristics of the selected learning algorithms for Markov models.

Algorithm	Model	Reference	Active	# states	Non-det.	JAJAPY	AALPY
BW-MC	MC	[11,12]	✗	✓	✓	✓	✗
BW-CTMC	CTMC	[13]	✗	✓	✓	✓	✗
MM-pCTMC	pCTMC	[13]	✗	✓	✓	✓	✗
BW-MDP	MDP	[11,12]	✗	✓	✓	✓	✗
Active-BW	MDP	[11,12]	✓	✓	✓	✓	✗
Alergia	MC	[6,7]	✗	✗	✗	✓	✓
IOAlergia	MDP	[8,9]	✗	✗	✗	✓	✓
L^*_{MDP}	MDP	[17]	✓	✗	✗	✗	✓

4 Architecture and Technical Aspects

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

Jajapy models. In JAJAPY, each kind of model is represented by a class, which inherits from an abstract class `Model`. This makes JAJAPY modular and easy to extend to other model formalisms.

The abstract class `Model` implements the methods to run the model, generate traces and compute the loglikelihood[1] of a set of traces under the model. Currently, all models in JAJAPY use an explicit state-space representation.

Every JAJAPY model has an attribute `matrix` which contains the transition probabilities. This `matrix` is a *Numpy ndarray* [39] of floats. Each JAJAPY model has also an attribute `labelling` containing the label associated to each state. This attribute is a Python list whose length equals to the number of states of the model. Finally, JAJAPY uses *Sympy* [40] to represent symbolic expressions used for transition rate expressions in pCTMCs.

[1] The logarithm of the likelihood function.

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.

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.

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 *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.

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.

In the experiments, we use the *loglikelihood distance* (abbreviated as *ll. distance*) as a metric to compare two models: given two models $\mathcal{M}$ and $\mathcal{M}'$ and n traces $\mathcal{O}$, the loglikelihood distance w.r.t. $\mathcal{O}$ is $\frac{1}{n}|\ln L(\mathcal{M},\mathcal{O}) - \ln L(\mathcal{M}',\mathcal{O})|$, where $L(\mathcal{M},\mathcal{O})$ denotes the likelihood of $\mathcal{O}$ under $\mathcal{M}$.

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 [41] 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.

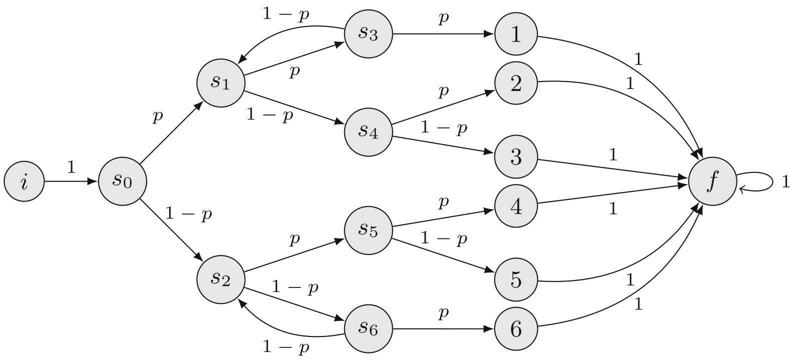

Fig. 3. The Yao-Knuth's die from [41]

We run these experiments on a Markov chain modelling the Yao-Knuth's die represented in Fig. 3 once with $p = 0.5$ (i.e. with a unbiased coin) and once with $p = 0.9$. Table 2 and 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 ll 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 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.

Table 2. Results for an unbiased Yao-Knuth's die ($p = 0.5$).

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

Table 3. Results for a biased Yao-Knuth's die ($p = 0.9$).

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

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. 4.

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

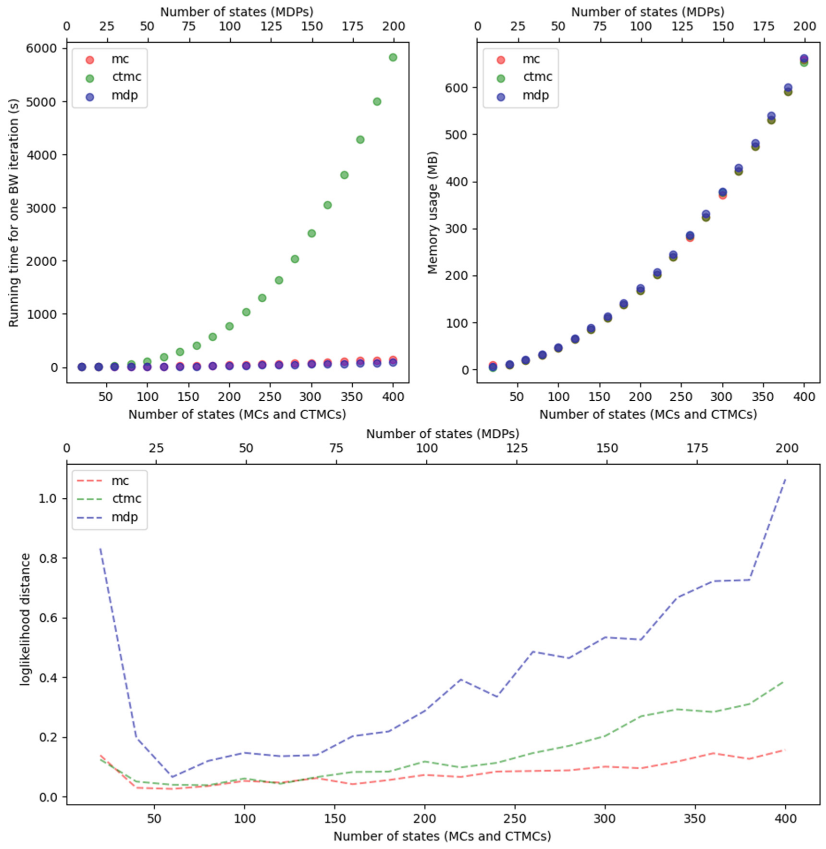

Fig. 4. JAJAPY running time, memory usage and loglikelihood distance w.r.t. the number of parameters of the hypothesis.

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.

Scalability Evaluation for pCTMCs. To evaluate the scalability of our software on pCTMCs, we use the tandem queueing network model from [42] (*cf.* Fig. 5) as a benchmark for our evaluation.

The experiments have been designed according to the following setup. We assume that the state of `serverC` is fully observable —i.e., its state variables `sc` and `ph` are– as well as the size `c` of the queue and the value of `lambda`. In contrast, we assume that the state of `serverM` is not observable.

Each experiment consists in estimating the value of the parameters `mu1a`, `mu1b`, `mu2`, and `kappa` from a training set consisting of 100 traces of length 30, generated by simulating the PRISM model depicted in Fig. 5. When the value of c is large, it is necessary to have lengthy traces to cover the state space of the SUL. As a result, traces with a length of 30 are utilised. However, in order to restrict the amount of time taken for execution, only 100 traces are employed. We perform this experiment both using timed and non-timed observations, by increasing the size c of the queue until the running time of the estimation exceeds a time-out set to 1 hour. We repeat each experiment 10 times by randomly re-sampling the initial values of each unknown parameter x_i in the interval $[0.1, 5.0]$. We annotate the running time as well as the relative error δ_i for each parameter x_i, calculated according to the formula $\delta_i = |e_i - r_i|/|r_i|$, where e_i and r_i are respectively the estimated value and the real value of x_i.

Figure 6 (bottom) depicts the graph of the average running time relative to the model size together with error bars. We observe that the running time is quadratic in the number of states (equivalently, linear in the size the number of states plus the number of non-zero transitions of the model) 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. In this respect, we observed that large models required more iterations than small models to converge. Nevertheless, all experiments required at most 20 iterations.

Figure 6 (top) details the average L_1-norm (resp. L_∞-norm) of the vector $\delta = (\delta_i)$, calculated as $\|\delta\|_1 = \sum_i |\delta_i|$ (resp. $\|\delta\|_\infty = \max_i |\delta_i|$). As one may expect, the variance of the measured relative errors is larger in the experiments performed with non-timed observations, and the quality of the estimation is better when employing 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 the parameters occur in many transitions.

```
ctmc
 // Tandem Queuing Network [Hermanns, Meyer-Kayser & Siegle]
const int c;  // queue capacity
const double lambda = 4 * c;
 // model parameters
const double mu1a = 0.2; const double mu1b = 1.8; const double mu2 = 2; const double kappa = 4;

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
```

Fig. 5. Prism model for the tandem queueing network from [42].

5.3 Comparison with AALpy

AALpy is an active automata-learning Python library that implements several learning algorithms to learn various families of automata. Since Jajapy learns stochastic models only, our comparison will focus on these models. However, AALpy is also able to Deterministic Finite Automata and Mealy Machines.

AALpy implements L^*_{MDP}, an extension of Angluin's L^* algorithm [17,18] to learn MDPs, and the Alergia algorithm to learn MCs.

Table 1 summarises the learning algorithms available in Jajapy and AALpy for stochastic models. On the one hand, Jajapy can learn non-deterministic models and CTMCs, which AALpy cannot; on the other hand, AALpy can learn non-stochastic models, which Jajapy cannot. In contrast to AALpy, Jajapy's output models are immediately usable in Stormpy.

We compare AALpy L^* and Jajapy BW algorithms on learning two variants of the grid-worlds presented in [12] and illustrated in Fig. 7. A robot is moving in the grid, starting from the top-left corner. Its objective is to reach the

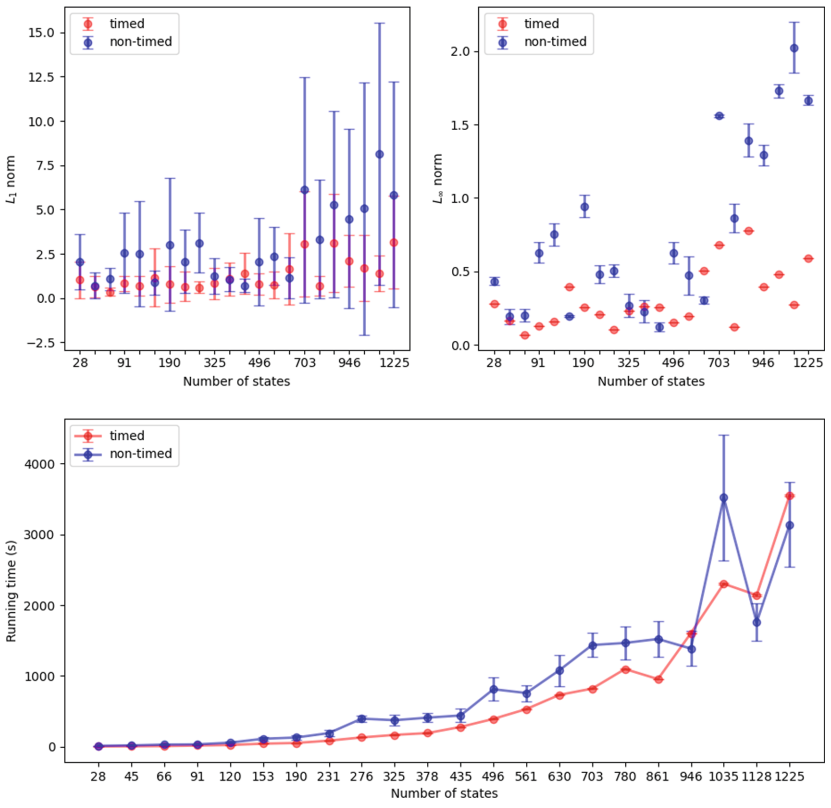

Fig. 6. Comparison of the performance of the estimation for timed and non-timed observations on the tandem queueing network with different size of the queue.

bottom-right corner. The actions are the four directions —north, east, south, and west— and the observed labels are the different terrains. Depending on the target terrain, the robot may make errors in movement, e.g. move southwest instead of south. By construction, the 3×3 world is a deterministic MDP, and the 4×4 world is a non-deterministic one.

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 4 and 5 show the results respectively for the 3×3 grid and the 4×4 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. [43] 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. [44] 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.

Table 4. Results for the 3×3 deterministic grid-world model.

	true	AALpy	Jajapy BW
overall # of labels	–	74, 285	74, 285
# of traces	-	15, 218	3, 714
$\lvert S \rvert$ (# of states)	17	18	**17**
loglikelihood distance	0.0	0.7305	**0.3352**
$\mathbb{P}_{\max}[F^{\leq 4}(\text{goal})]$	0.336	0.322	**0.347**
$\mathbb{P}_{\max}[\neg \text{G}\ U^{\leq 4}(\text{goal})]$	0.072	0.074	0.074
Running time	–	**1.15 s**	290.8 s

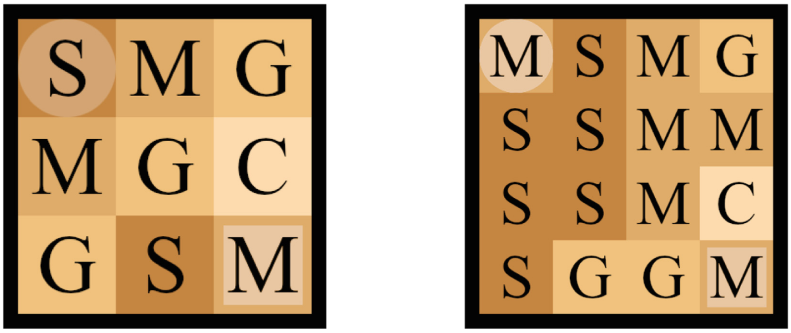

Fig. 7. Grid worlds models. (Left) a 3×3 deterministic model; (Right) a 4×4 non-deterministic model.

Table 5. Results for the 4×4 non-deterministic grid-world model.

	true	AALpy	Jajapy BW
overall # of labels	–	16,232,244	**200,000**
# of traces	–	$2,174,167$	**10,000**
$\lvert S \rvert$ (# of states)	28	207	**28**
loglikelihood distance	0.0	0.4963	**0.4680**
$\mathbb{P}_{\max}[F^{\leq 7}(\text{goal})]$	0.687	0.680	**0.692**
$\mathbb{P}_{\max}[F^{\leq 12}(\text{goal})]$	0.996	0.995	**0.996**
$\mathbb{P}_{\max}[\neg(\text{C} \mid \text{W})\ U^{\leq 7}(\text{goal})]$	0.520	**0.514**	0.504
Running time	–	**290.65 s**	15,303.83 s

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 [45,46].

Data Availability. An artifact allowing one to reproduce the experiments from this paper has been submitted to the QEST 2023 artifact evaluation.

References

1. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
2. Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T., Volk, M.: The probabilistic model checker storm. Int. J. Softw. Tools Technol. Transfer **24**, 1–22 (2022)
3. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst. Des. **15**(1), 7–48 (1999). https://doi.org/10.1023/A:1008739929481
4. Budde, C.E., et al.: On correctness, precision, and performance in quantitative verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12479, pp. 216–241. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-83723-5_15
5. Reynouard, R.: Jajapy github repository (2022). https://github.com/Rapfff/jajapy
6. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Carrasco, R.C., Oncina, J. (eds.) ICGI 1994. LNCS, vol. 862, pp. 139–152. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58473-0_144
7. Carrasco, R.C., Oncina, J.: Learning deterministic regular grammars from stochastic samples in polynomial time. RAIRO - Theor. Inf. Appl. (RAIRO: ITA) **33**(1), 1–20 (1999)
8. Mao, H., Chen, Y., Jaeger, M., Nielsen, T., Larsen, K., Nielsen, B.: Learning probabilistic automata for model checking, pp. 111–120, October 2011
9. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning deterministic probabilistic automata from a model checking perspective. Mach. Learn. **105**(2), 255–299 (2016)
10. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proc. IEEE **77**(2), 257–286 (1989)
11. Bacci, G., Ingólfsdóttir, A., Larsen, K.G., Reynouard, R.: Active learning of markov decision processes using baum-welch algorithm (extended), CoRR, vol. abs/2110.03014 (2021). https://arxiv.org/abs/2110.03014
12. Bacci, G., Ingólfsdóttir, A., Larsen, K.G., Reynouard, R.: Active learning of markov decision processes using baum-welch algorithm. In: 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1203–1208 (2021)
13. Bacci, G.: Mm algorithms to estimate parameters in continuous-time markov chains (2023). https://arxiv.org/abs/2302.08588
14. Reynouard, R.: Jajapy's documentation (2022). https://jajapy.readthedocs.io/en/latest/
15. Reynouard, R.: A short introduction to jajapy, March 2023. https://doi.org/10.5281/zenodo.7695105
16. Muškardin, E., Aichernig, B., Pill, I., Pferscher, A., Tappler, M.: Aalpy: an active automata learning library. Innov. Syst. Softw. Eng. **18**, 1–10 (2022)
17. Tappler, M., Muškardin, E., Aichernig, B.K., Pill, I.: Active model learning of stochastic reactive systems. In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021. LNCS, vol. 13085, pp. 481–500. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_27
18. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
19. Biere, A., Bloem, R. (eds.): CAV 2014. LNCS, vol. 8559. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9

20. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: The automata learning framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_32
21. Thollard, F., Dupont, P., de la Higuera, C.: Probabilistic DFA inference using kullback-leibler divergence and minimality. In: International Conference on Machine Learning (2000)
22. Stolcke, A.: Bayesian learning of probabilistic language models (1994)
23. Gelfand, A., Smith, A.: Sampling-based approaches to calculate marginal densities. J. Am. Stat. Assoc. **85**, 398–409 (1990)
24. Neal, R.: Markov chain sampling methods for dirichlet process mixture models. J. Comput. Graph. Stat. **9**, 249–265 (2000)
25. Rashidinejad, P., Zhu, B., Ma, C., Jiao, J., Russell, S.J.: Bridging offline reinforcement learning and imitation learning: a tale of pessimism. IEEE Trans. Inf. Theor. **68**, 8156–8196 (2021)
26. Jin, Y., Yang, Z., Wang, Z.: Is pessimism provably efficient for offline rl? In: International Conference on Machine Learning (2020)
27. Buckman, J., Gelada, C., Bellemare, M.G.: The importance of pessimism in fixed-dataset policy optimization, ArXiv, vol. abs/2009.06799 (2020)
28. Morimoto, J., Doya, K.: Robust reinforcement learning. Neural Comput.**17**(2), 335–359 (2005). https://doi.org/10.1162/0899766053011528
29. Lim, S.H., Xu, H., Mannor, S.: Reinforcement learning in robust markov decision processes. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. (eds.), Advances in Neural Information Processing Systems, vol. 26. Curran Associates Inc, (2013). https://proceedings.neurips.cc/paper_files/paper/2013/file/0deb1c54814305ca9ad266f53bc82511-Paper.pdf
30. Suilen, M., Simão, T.D., Parker, D., Jansen, N.: Robust anytime learning of markov decision processes (2023)
31. Derman, E., Mankowitz, D.J., Mann, T.A., Mannor, S.: A bayesian approach to robust reinforcement learning. In: Conference on Uncertainty in Artificial Intelligence (2019)
32. Češka, M., Hensel, C., Junges, S., Katoen, J.-P.: Counterexample-guided inductive synthesis for probabilistic systems. Form. Asp. Comput. **33**(4–5), 637–667 (2021). https://doi.org/10.1007/s00165-021-00547-2
33. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.: Parameter synthesis for markov models: faster than ever, CoRR, vol. abs/1602.05113 (2016). http://arxiv.org/abs/1602.05113
34. Jansen, N., Junges, S., Katoen, J.: Parameter synthesis in markov models: a gentle survey. In: Raskin, J., Chatterjee, K., Doyen, L., Majumdar, R. (eds.), Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday, ser. LNCS, vol. 13660, pp. 407–437. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-22337-2_20
35. Polgreen, E., Wijesuriya, V.B., Haesaert, S., Abate, A.: Data-efficient bayesian verification of parametric markov chains. In: Agha, G., Van Houdt, B. (eds.) QEST 2016. LNCS, vol. 9826, pp. 35–51. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43425-4_3
36. Baum, L., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains (1970)
37. Yang, F., Balakrishnan, S., Wainwright, M.J.: Statistical and computational guarantees for the baum-welch algorithm. J. Mach. Learn. Res. **18**(125), 1–53 (2017). http://jmlr.org/papers/v18/16-093.html

38. Hoeffding, W.: Probability inequalities for sum of bounded random variables (1961)
39. Harris, C.R., et al.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2
40. Meurer, A., et al.: Sympy: symbolic computing in python. PeerJ Comput. Sci. **3**, e103 (2017). https://doi.org/10.7717/peerj-cs.103
41. Knuth, D., Yao, A.: Algorithms and Complexity: New Directions and Recent Results. Academic Press, Cambridge (1976), ch. The complexity of nonuniform random number generation
42. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In: Plateau, B., Stewart, W., Silva, M. (eds.) Proceedings of 3rd International Workshop on Numerical Solution of Markov Chains (NSMC 1999), Prensas Universitarias de Zaragoza, pp. 188–207 (1999)
43. Khreich, W., Granger, E., Miri, A., Sabourin, R.: On the memory complexity of the forward-backward algorithm. Pattern Recogn. Lett. **31**(2), 91–99 (2010). https://www.sciencedirect.com/science/article/pii/S0167865509002578
44. Bartolucci, F., Pandolfi, S.: Comment on the paper on the memory complexity of the forward-backward algorithm. In: khreich, W., Granger, E., Miri, A., Sabourin, R. (eds.), Pattern Recognition Letters, vol. 38, pp. 15–19 (2014). https://www.sciencedirect.com/science/article/pii/S0167865513003863
45. Shao, Y., Wang, Y., Povey, D., Khudanpur, S.: Pychain: a fully parallelized pytorch implementation of LF-MMI for end-to-end ASR. In: Meng, H., Xu, B., Zheng, T.F. (eds.), Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25–29 October 2020, ISCA, pp. 561–565 (2020). https://doi.org/10.21437/Interspeech.2020-3053
46. Ondel, L., Lam-Yee-Mui, L., Kocour, M., Corro, C.F., Burget, L.: Gpu-accelerated forward-backward algorithm with application to lattice-free MMI. In: IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2022, Virtual and Singapore, 23–27 May 2022, pp. 8417–8421. IEEE (2022). https://doi.org/10.1109/ICASSP43922.2022.9746824