Seminarpaper

Mario Wagner, 0730223

Graz, am November 9, 2016

Contents

List of Figures

List of Tables

1 Introduction

Supervised learning is the task of inferring knowledge from a fully labeled training data. In contrast, unsupervised learning is the task of inferring knowledge from an unlabeled dataset. The common ground of supervised and unsupervised learning, is the task of inferring knowledge from a small amount of labeled data and a huge amount of unlabeled training data; i.e., *semi-supervised learning*. In other words, semi-supervised learning is making use of that small amount of data (i.e., supervised learning) to do unsupervised learning considerably more accurate.

Active learning is a special case of semi-supervised learning in which the labeled data is incrementally provided by a continuous interaction between the learner and an *Oracle*. The oracle can be a user, a system, or a teacher that provides a certain level of insight to supervise the learner by letting it to actively query for labels of unknown data. When the oracle is a system, actively querying it only means the learner is iteratively providing the system with input alphabet and asking for the output alphabet.

This method is commonly used to learn models of reactive systems by actively providing them with external events (i.e., inputs) and observe how they react to them (i.e., outputs). Though, sometimes active learning can be as expensive as manual labeling since not all runs of all systems are inexpensive or repeatable; to exemplify, consider a scenario in which an input sequence triggers a safety critical behaviour of the system (e.g. self-termination).

A popular algorithm proposed by Angluin[1] is called L^*. A so called *Learner* efficiently learns an unknown regular set from a *Teacher* by repeatedly asking questions and saving the findings in an observation table. There are two types of questions the *Learner* can ask the *Teacher*. First, it can ask whether a given string is a member of the unknown set (membership query) and the *Teacher* replies with yes or no. Second, a conjecture can be made and the *Teacher* answers if the set is correct (equivalence query). If the set is incorrect, a counterexample is provided. The *Teacher* can also be substituted by a random sampling *Oracle*. The outcome of the algorithm is a minimal deterministic finite automaton (DFA).

Although L^* is a quite useful automata learning approach, there are also issues with it; first of which is while using L^* the learner either learns the correct system or nothing. Next issue with L^* is actually stated earlier, that is, when it is impossible for some systems to perform some queries for various reasons (e.g. not being cost-effective, safety critical behaviour, expensive overheads, and etc.). Because of these reasons and many others that we may not know beforehand, passive learning (i.e., unsupervised learning) is used to learn the somewhat correct model of the system under learn. AALERGIA[2] is an interesting approach to learn stochastic automata from a data-set of traces of a system.

In the industry, model-driven development techniques are getting more and more important, as you can use them for simulation, documentation, model-checking, performance evaluation and many other things. The main problem with these techniques is the part where the model is constructed, because it is often costly and time consuming. AALERGIA tries to ease this by automatically constructing a high level model from

traces of an unknown system.

There are some difficulties when trying to learn deterministic models, which can be overcome by learning probabilistic models. One of the main problems is that the behaviour of a system is not always deterministic (for example, different physical environments) and includes noise. AALERGIA learns a probabilistic model by using only previously observed patterns, thus, the system cannot be queried as in Angluin's algorithm (no active learning). In contrast to statistical model checking, the algorithm extracts an explicit probabilistic model, which one can use to verify a large class of properties without the need to re-sample the system.

It is crucial to note that learning these models can represent different views of the underlying system, depending on the input and output actions. Therefore, if we tailor the observations to the properties of interest, our verification will be simpler and more efficient.

2 Preliminaries

2.1 Markov Models

Markov model is a stochastic model capable of depicting randomly changing systems assuming Markov property. Markov property states future state is only dependant on the current state and not on the events that occurred before. There are different types of Markovian models each of which is used in different scenarios.

Markov chain is the simplest variation of Markovian models used to model autonomous systems whose states are fully observable. Here we study Deterministic Labeled Markov Chain (DLMC) and Discrete-Time Markov Chain (DTMC). Meanwhile Markov Decision Process is another variation of Markovian models, used to model non-autonomous (controlled) systems whose states are fully observable, which we study in this section.

2.1.1 Deterministic Labeled Markov Chain

A labeled Markov chain is a tuple $M = (Q, AP, \pi, \tau, L)$ where Q is a finite non-empty set of states, AP a finite non-empty set of atomic propositions, $\Sigma = 2^{AP}$ is a finite alphabet, $L : Q \to \Sigma$ is a state labeling function, and π is the initial probability distribution as follows

$$\pi : Q \to [0, 1] \text{ s.t. } \sum_{q \in Q} \pi(q) = 1,$$

and τ is the transition probability function as follows:

$$\tau : Q \times Q \to [0, 1] \text{ s.t. } \forall q \in Q \cdot \sum_{q' \in Q} \tau(q, q') = 1.$$

A labeled Markov chain is deterministic (DLMC) if (1) there exist an initial state $q_0 \in Q$ such that $\pi(q_0) = 1$, and (2) for all $\sigma \in \Sigma$ and from all states $q \in Q$ there exists at most one destination state labeled with σ that is $\tau(q, q') > 0$ and $L(q') = \sigma$.

2.1.2 Discrete-Time Markov Chain

A Discrete-Time Markov Chain (DTMC) is a tuple $M = (Q, q_0, AP, \tau, L)$ where Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, AP a finite non-empty set of atomic propositions, $\Sigma = 2^{AP}$ is a finite alphabet, $L : Q \to \Sigma$ is a state labeling function, and τ is the transition probability function as follows:

$$\tau : Q \times Q \to [0, 1] \text{ s.t. } \forall q \in Q \cdot \sum_{q' \in Q} \tau(q, q') = 1.$$

DTMC is actually representing a stochastic processes in discrete time $n \in \mathbb{N}$ as a sequence of random variables $Q = q_n$. We refer to q_n as the state of process at time n, with q_0 denoting the initial state. If the random variable q_i takes values in discrete space such as integers $\mathbb{Z}$, then the we are dealing with a discrete-valued stochastic process.

2.1.3 Markov Decision Procedure

A Markov Decision Procedure (MDP) is an extended DTMC such that it allows both probabilistic and non-deterministic behaviour. MDP is a tuple $M = (Q, q_0, AP, \tau, L)$ where Q is a finite non-empty set of states, q_0 is the initial state, AP is a finite non-empty set of atomic propositions, $\Sigma = 2^{AP}$ is a finite alphabet, $L : Q \to \Sigma$ is a state labeling function, and $\tau : Q \times AP \times Q \to [0, 1]$ is the partial probabilistic transition function defined for all q and σ where $\sigma \in L(q)$.

2.2 PRISM Model Checker

Dealing with deterministic models it is easy to associate states with strings. Despite the fact that it is not a one-to-one association it is an effective lexicographical representation for not only it is possible to identify a particular state by having a string belonging to set of all strings emanates from it but it also is possible to denote an arbitrary string by a particular state. A path over the states $q_0, \ldots, q_n$ of a state labeled deterministic Markovian model M gives rise to a trace/string $s = \sigma_1, \ldots, s_n$ whose probability of occurrence is defined by $\mathcal{P}_{\mathrm{M}}(s)$ as:

$$\mathcal{P}_{\mathrm{M}}(s) = \prod_{i=0}^{n-1} \tau(q_i, q_{i+1}),$$

Meanwhile, for non-deterministic Markovian models such as MDP it is also possible to perform Monte-Carlo simulation to estimate the probability of a trace occurrence.

PRISM model checker harnesses above described approaches to verify the correctness of stochastic properties of randomly changing systems over their stochastic models. These stochastic properties can be formally formulated by Probabilistic LTL which we are going to study in the next subsection.

2.2.1 Probabilistic Linear Temporal Logic

Modal and temporal properties of Markovian models are difficult to formulate because they represent randomly changing systems. Meanwhile, LTL is a modal temporal logic best suited to formally express the properties of reactive systems. In this section, we study a probability extended LTL that is suited to formulate the properties of Markovian models.

Linear temporal logic over set of atomic propositions AP is defined by the following syntax:

$$\varphi ::= \mathbf{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \mathbf{G}\,\varphi \mid \mathbf{F}\,\varphi \mid \varphi_1\,\mathbf{U}\,\varphi_2 \qquad (a \in AP)$$

The syntax of Probabilistic LTL (PLTL) is defined as:

$$\phi ::= \mathcal{P}_{\bowtie \mathrm{r}}(\varphi)$$

where φ is an LTL property, $\bowtie$ is a comparison operator $\{\leq, <, =, \geq, >\}$ and $r \in [0, 1]$ is a confidence threshold. Semantic of a PLTL can be defined over a deterministic Markovian model M as:

$$M \models \mathcal{P}_{\bowtie r}(\varphi) \text{ iff } \mathcal{P}_{\mathrm{M}}(\varphi) \bowtie r,$$

where $\mathcal{P}_{\mathrm{M}}(\varphi)$ denotes $\mathcal{P}_{\mathrm{M}}(\{s \mid s \models \varphi\})$ meaning a string belonging to M models φ.

2.2.2 Dice Model

The first and simpler example we are going to use is a PRISM model of a simple probabilistic algorithm, the dice model[3]. Figure 1 shows the states and the probability of the state transitions. We start at an initial state 0 and we toss a coin at each step, giving us a 50 percent chance of transiting to each of the two following states. The algorithm ends at the leafs of the tree, which represent the values of the dice (d1-d6).

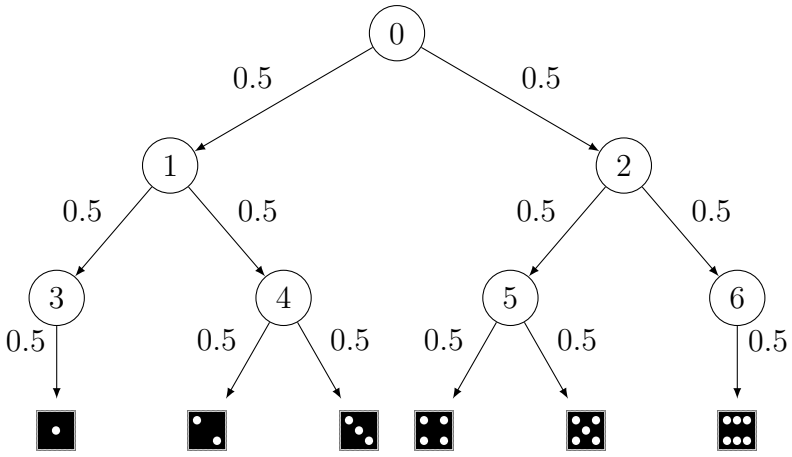

Figure 1: The Dice Example

The source code of the dice PRISM model is shown in figure 2. At the beginning, the code indicates that it is describing a discrete-time Markov chain (DTMC). The source code shows a single PRISM module called dice. The variable s, which represents the state of the system, can take the values from 0 to 7 and is initialised with the value 0 (line 6). The variable d, which represents the value of the dice, can range from 0 to 6 and is also initialised with the value 0, meaning that the dice has no value yet (line 8). Each line depicts a state and the probabilities of the transitions, e.g. line 11 shows there is a probability of 0.5 of transiting to state 3 or state 4 if we are currently in state 1. The model ends if a value is assigned to the dice, as there are no further transitions from that point on.

2.2.3 Herman's Self-Stabilisation Algorithm

The running example we are going to use in this paper (unless explicitly stated otherwise) is a PRISM model of Herman's self-stabilising algorithm, as described in [4]. The

```
dtmc

module dice

  // local state
  s : [0..7] init 0;
  // value of the dice
  d : [0..6] init 0;

  [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
  [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
  [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
  [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
  [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
  [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
  [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
  [] s=7 -> (s'=7);

endmodule
```

Figure 2: The Prism Dice Model

algorithm guarantees that the system recovers from faults in a certain amount of time. Although it is certain that the system self-stabilises eventually, it is very hard to analyse the maximum execution time, which depicts the worst case scenario in this case.

Herman's algorithm works with a network of an odd number of identical processes in a ring, ordered anticlockwise. The processes work synchronously and possess a token, which is passed around the ring. The result of a random coin toss decides whether the process keeps the token or passes it on to its neighbour. If a process holds two tokens, they are eliminated. The result is a stable system, where each process has exactly one token and the tokens are passed around forever in the ring. In order to analyse the correctness and the performance of the proposed model, the authors use the tool PRISM Model Checker and invoke probabilistic model checking methods on the model.

The proposed methods worked well and proved the correctness of the algorithm. Their findings on the maximum execution time showed, that having 3 tokens, spaced evenly around the ring, always produced the worst-case behaviour. It is stated that probabilistic model checking is extremely useful for checking models of this kind, because simple tools, languages and techniques are provided to model the system. Also, an exhaustive analysis can be done, as opposed to other simulation techniques such as the Monte Carlo simulation. A disadvantage is the restriction of probabilistic model checking to finite state models. Furthermore, the size of the models under analysis are limited to the amount of time and space available.

The source code of the model is shown in figure 3. In this case, seven processes are used in the ring. The step notation in line 8 forces all processes to synchronise. The state of each process is depicted by a single two-valued variable (x1-x3). Initially, the value is 1 for all processes. The implementation of this model is straightforward and it

is used as a data generator for the input of the AALERGIA implementation.

```
dtmc

// module for process 1
module process1

  x1 : [0..1] init 1;

  [step] (x1=x3) -> 0.5 : (x1'=0) + 0.5 : (x1'=1);
  [step] !(x1=x3) -> (x1'=x3);

endmodule

// add further processes through renaming
module process2 = process1[x1=x2, x3=x1 ] endmodule
module process3 = process1[x1=x3, x3=x2 ] endmodule
```

Figure 3: The Prism Herman 3 Model

2.3 AALERGIA

ALERGIA[5] was introduced by Rafael C. Carrasco and Jose Oncina in 1994. They proposed an algorithm which is capable of identifying any stochastic deterministic regular language by using a prefix tree acceptor and merging the states in $O(|S|^3)$, where $|S|$ is the size of the sample. This is especially useful for learning automaton in realistic situations. They based their approach on previous work from Oncina[6]. ALERGIA has been extended in many ways, one of many is called AALERGIA[2].

The reason for creating AALERGIA was to show that the methods for learning probabilistic automata can be adapted for learning Markov Chain system models for verification. The algorithm also leads to stronger consistency for learning in the limit than previous implementations and the authors analyse how the convergence of the learned model relates to the convergence of probability estimates for system properties. This enables us to use the learned model for probabilistic linear temporal logic (PLTL) model checking.

However, there is still room for improvement, as shown in a competition called PAutomaC[1] by the winner team[7]. They introduced a new criterion for merging states based on marginal probability by greedily merging states if it increases the probability of the automaton.

In this paper, we take AALERGIA and the provided MATLAB implementation[2], implement it in python and experiment with it. In order to get a better idea of the way the implementation works, we will describe it in detail and a running example for

[1] hai.cs.umbc.edu/icgi2012/challenge/Pautomac/index.php
[2] http://mi.cs.aau.dk/code/aalergia

better replicability is used throughout this section. The implementation of the algorithm consists of several parts, which are depicted in the figure 4.

Figure 4: The Flow Diagram AALERGIA

2.3.1 Training Data

The data used to learn the Deterministic Labeled Markov Chain (DLMC) is supplied in the form of a comma separated values file, which contains the training set and the alphabet to be used. The alphabet consists of a finite 1×N-cell array, where each column contains one letter in the alphabet. The training set consist of a finite N×1-cell array, where each row contains sequences of symbols generated from the model, separated by commas.

Dataset Generator It is easy to generate datasets by using the command line version of PRISM. It is possible to generate random paths through a model by specifying the model file, an output file and the path to be generated (for instance, how many steps should be processed). The output file can then be used as input file for the AALERGIA implementation.

The running example we are going to use is a self-stabilizing ring network with 3 processes. Table 1 shows the beginning of the training set, generated by the PRISM modelchecker, which contains 50 Sequences in total. In our case, each symbol in the training set represents the states in the ring at a given moment. For example, the symbol 001 shows that process 3 is in the state 1 and process 1 and 2 are in the state 0.

	1
1	000,001,
2	000,011,100,010,101,010,001,100,010,101,
3	000,101,
4	000,111,010,001,110,011,101,010,001,110,011,
5	000,111,011,101,110,001,
6	000,101,010,101,110,011,100,011,101,110,

Table 1: The Training set

Table 2 shows the alphabet, which consists of 8 symbols from 000 to 111.

	1	2	3	4	5	6	7	8
1	000	001	010	011	100	101	110	111

Table 2: The Alphabet

2.3.2 Frequency Prefix Tree Acceptor

After loading the training set and the alphabet into the workspace, the Frequency Prefix Tree Acceptor (FPTA) is created. A deterministic frequency finite automaton (DFFA) is a tuple $A = \langle \Sigma, Q, I_{\text{fr}}, F_{\text{fr}}, \delta_{\text{fr}}, \delta \rangle$ where Σ = the finite alphabet, Q = the finite set of states, I_{fr} = the initial state frequencies, F_{fr} = the final state frequencies, δ_{fr} = the frequency transition function and δ = the transition function.

Figure 5 shows how a simplified FPTA for the in section 2.2.2 described dice model looks like. The sample S is a multiset of size 20. S = {(SH,1), (SHHH,1), (SHHHD_1,1), (SHHD_1,2), (SHHD_1D_1,1), (SHT,2), (SHTD_2,1), (SHTD_2D_2,1), (SHTD_3,1), (ST,1), (STH,2), (STHD_4,1), (STHD_5,1), (STT,2), (STTD_6,2)}. The letter H stands for heads, the letter T stands for tails and D1 - D6 stands for the different dice values. The numbers in the state indicates the number of the occurrence of the event, whereas the number on the transitions illustrates the count of all events below it.

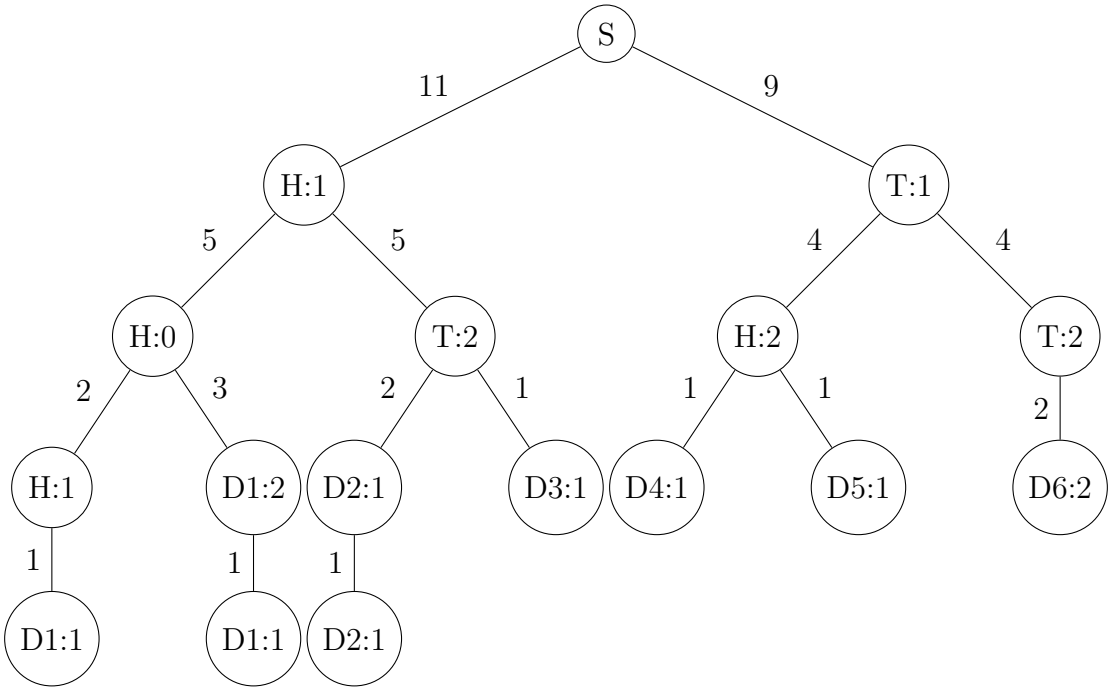

Figure 5: The FPTA for the DICE model

Algorithm 1 shows how the creation of the FPTA is implemented in the AALERGIA package. The code starts by sorting the training set and by removing double occurrences (line 3). After that, a for-loop iterates through the sorted strings and splits them at each

comma (line 6 - 15). This creates all the prefixes from the training set and adds them to the cell array named *prefix*, as shown in Table 3.

	1
1	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,001,
2	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,
3	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,
4	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,
5	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,
6	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,
7	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,
8	000,000,010,001,100,011,101,110,001,110,001,110,011,101,
9	000,000,010,001,100,011,101,110,001,110,001,110,011,
10	000,000,010,001,100,011,101,110,001,110,001,110,
11	000,000,010,001,100,011,101,110,001,110,001,

Table 3: The Prefixes

In our example, the dimension of *prefix* is 494x1 and contains all unique prefixes of the training set for further evaluation. The cell array *prefix* is sorted by width-first-sort (line 16). The shortest strings are at the beginning of the array and the largest at the end of the array, as shown in Table 4.

	1
1	000,
2	000,000,
3	000,001,
4	000,010,
5	000,011,
6	000,100,
7	000,101,
8	000,110,
9	000,111,
10	000,000,010,
11	000,000,100,
12	000,000,111,
13	000,001,100,
14	000,001,110,

Table 4: The Sorted Prefix Array

The function *find_predecessor* searches through the *prefixes* array and saves all the predecessors in the corresponding array (line 16). In the end, a for-loop over all the prefixes is executed in order to find all transitions and their frequencies respectively.

Algorithm 1 Create the FPTA

Input: the $\sum$ and a set of strings S (training data)
Output: a DFFA A and a sorted set of strings U
$U \leftarrow \text{sort}(S)$
prefixes $\leftarrow U$
$F_{fr} \leftarrow \text{string\_count}(U)$
for `u` $\in$ `U` **do**
    index $\leftarrow$ find positions of "," in u
    while *index* > 0 **do**
        substr $\leftarrow$ *substring(1:index)*
        if *substr* $\notin$ *prefix* **then**
            prefixes(end+1) $\leftarrow$ *substr*
        end if
        index $\leftarrow$ *index-1*
    end while
end for
prefixes $\leftarrow$ width_first_sort(*prefixes*)
predecessor $\leftarrow$ find_predecessor(*prefixes*)
for `p` $\in$ `prefixes` **do**
    sym $\leftarrow$ *get_symbol(p)*
    pre $\leftarrow$ *predecessor(p)*
    freq_trans_matrix^1 *(pre, sym)* $\leftarrow$ *predecessor(p)*
    frequency_transition $\leftarrow$ *freq_trans_matrix*^{2} *(pre, sym)*
    frequency_transition $\leftarrow$ *frequency_transition* + *frequency(p)*
    while *pre* **do**
        update_freq_trans_matrix(*p, pre*)
        pre $\leftarrow$ *predecessor(p)*
    end while
end for
init_states $\leftarrow$ *freq_trans_matrix*^{1}*(1, all)*
return A

The values are saved into the *freq_trans_matrix* (line 18 - 28). This Matrix contains 2 elements: the first element is the transition matrix, which contains all the transition between the states. The second element is a matrix, which contains all the frequencies between the states.

The returned object DFFA (line 30) has the following members:

- a finite set of states
- the labels of the states
- the alphabet
- the initial state
- the initial state frequency
- the final state frequency
- the frequency transition matrix
- RED (later used for merging) [* **ME**: be more comprehensive about RED set *]
- BLUE (later used for merging) [* **ME**: likewise *]

Table 5 shows how the first element of the frequency transition matrix (the transition matrix) is structured. The columns represent the 8 symbols, the rows the different states. For example, row number 2, column 1 shows the number 3. This indicates that state 2 and symbol 1 lead to state 3. The second element of the frequency transition matrix has the same dimensions as the first element and follows the same logic, but instead of transitions it contains the frequencies.

	1	2	3	4	5	6	7	8
1	2	0	0	0	0	0	0	0
2	3	4	5	6	7	8	9	10
3	0	0	11	0	12	0	0	13
4	0	0	0	0	14	0	15	0
5	0	16	0	0	0	17	0	0

Table 5: The Transition Matrix

2.3.3 Normalization

The algorithm 2 shows how a DFFA is converted into a deterministic probabilistic finite automaton (DPFA). The computation is pretty straightforward. As input, the algorithm takes a well defined DFFA, that means that the number of strings entering and leaving a given state is identical. The probabilistic transition matrix (line 5) is identical to

the frequency transition matrix, but its second element contains the probabilities of the transitions instead of the frequencies of the transitions.

The algorithm searches through the frequency transition matrix, sums up the frequencies of the transitions and adds them to the frequency of the state itself in order to get the total number of frequencies (line 7). Following that, the frequency of each transition and of the state is divided by the total number of total frequencies, which results in the corresponding probabilities for the transitions. (line 9).

Algorithm 2 Create a DPFA from a DFFA

```
1:  Input: a well defined DFFA
2:  Output: corresponding DPFA
3:  for q ∈ Q do
4:      freq_state ← dffa.finalStateFrequency(q)
5:      ptm^1 ← dffa.frequencyTransitionMatrix^1
6:      freq_trans ← dffa.frequencyTransitionMatrix^2(q, all)
7:      freq_total ← freq_trans + freq_state
8:      if freq_total > 0 then
9:          ptm^2(node, index) ← freq_trans / freq_total
10:         fsp(q) ← freq_state / freq_total
11:     end if
12: end for
13: return DPFA(dffa, fsp, ptm)
```

2.3.4 Golden Section Search

First, we need to search for a left and right border of ε in order to get a region for our golden section search. This is done by the following algorithm 3. In order to save space, the algorithm only explains how to find the left border of the region. The right border is found the same way, with some minor modifications. The algorithm iteratively calculates BIC-scores for the values of ε. For each run of the while-loop, the ε is multiplied by 0.5 until a left border is found. If the new ε produces a Bayesian Information Criterion(BIC)-score larger than the previous ε (line 8), it essentially means that we found a right border, but the left border is further to the left and we need to search again. If the new score is the same as the old score (line 13), we search again. If the value of the score doesn't change for 5 consecutive tries, we break the loop and take the last value of ε as our left border. Last but not least, if the new score is smaller than the old score (line 17), we found the left border and can continue searching for the right border (not shown in the pseudo code).

BIC Score This is a measure that can be used to evaluate the learned model. It creates a score that is calculated on the likelihood minus a penalty term for the model complexity. The BIC score of a given DLMC A given data $S[n]$ is defined in formula 1

$$BIC(A|S[n]) := log(P_A(S[n])) - 1/2|A|log(N) \quad (1)$$

In this formula, $|A|$ depicts the size of A and $N = \sum l_i$ is the total size of the data.

Algorithm 3 Find ε-region

```
Input: the DFFA
Output: the region for ε, scores, epsilon_values
ε ← 1
score ← calculate_BIC_Score(ε, dffa)
new_ε ← ε * 0.5
while new_ε > 0 do
    new_score ← calculate_BIC_score(new_ε, dffa)
    if new_score > score then
        region_right ← ε
        ε ← new_ε
        new_ε ← new_ε * 0.5
        score ← new_score
    else if new_score = score then                ▷ do this max. 5 times
        ε ← new_ε
        new_ε ← new_ε * 0.5
        score ← new_score
    else
        region_left ← new_ε
        break
    end if
end while
...
...
return region_right, region_left
```

Table 6 shows the values of the left and right border of ε. The algorithm ran only once for the left border (ε startes with 1 and is halfed every run) and 5 times for the right border, since the scores always stayed the same. Table 7 shows the BIC scores and table 8 shows the corresponding ε values. For example, we calculated the BIC score for $\varepsilon = 0.5$ and the result was $-1.5769e + 03$.

	1	2
1	0.5	64

Table 6: The Search Region

After having found the region, the golden section search can begin. Algorithm 4 shows how it is implemented. Basically, the algorithm uses the golden ratio on our 2 borders

	1	2	3	4	5	6	7	8
1	-840.608	-1.5769e+03	-840.608	-840.608	-840.608	-840.608	-840.608	-840.608

Table 7: The BIC scores

	1	2	3	4	5	6	7	8
1	1	0.5	2	4	8	16	32	64

Table 8: The Epsilon values

a1 and a2 and tries to find a maximum (the highest BIC score) between these 2 values. If it exists, the section that contains the maximum is selected as new region and the search is started again, until the recursion reaches depth 3 or the condition at line 11 is fulfilled. At line 7, 8 we calculate the 2 values a1,a2 for our golden ratio, given the region r1,r2. After that, the corresponding BIC scores f1,f2 are calculated(line 9, 10). If the distance between the 2 values a1,a2 is small enough, we stop the recursion and the average score between the 2 values is taken as a good ε (line 12). If f1 is smaller than f2, the maximum has to be between a1 and r2 and a new golden section search is started. If it is the other way around and f1 is larger than f2, the maximum is between r1 and a2 and we start the search with these values. If it happens that the score of f1 and f2 is exactly the same, we need to make sure, that the recursion is not endless (line 24). We check if f1 is larger than the largest score we already calculated. Should this be the case, the maximum is between a1 and a2 (line 27). Lastly, if we do not have any luck and we have not found a maximum so far, we simply start 2 new golden searches for the left (r1, a1) and the right (a2, r2) region respectively (line 34, 35). In the end, the algorithm returns a suggestion for a good ε as shown in Table 9, which we use as input parameter (α) for our merging algorithm.

	1
1	64

Table 9: The Epsilon

Algorithm 4 Golden_section_search

```
1: Input: region, dffa, scores
2: Output: good_eps, scores, epsilon
3: good_eps ← 0
4: r1 ← region(1,1)
5: r2 ← region(1,2)
6: while true do
7:     a1 ← r1 + 0.382(r2-r1)
8:     a2 ← r2 + 0.618(r2-r1)
9:     f1 ← calculate_BIC_score(a1,dffa)
10:    f2 ← calculate_BIC_score(a2, dffa)
11:    if |a1-a2| < 0,00001 then
12:        good_eps ← (a1 + a2) / 2
13:        return
14:    end if
15:    if f1 < f2 then
16:        region ← region(a1, r2)
17:        good_eps, scores, epsilon ← Golden_section_search(region, dffa, scores)
18:        return
19:    else if f1 > f2 then
20:        region ← region(r1, a2)
21:        good_eps, scores, epsilon ← Golden_section_search(region, dffa, scores)
22:        return
23:    else
24:        if recursion_depth = 3 then
25:            return
26:        end if
27:        if f1 > max(scores) then
28:            new_region ← a1, a2)
29:            good_eps, scores, epsilon ← Golden_section_search(new_region, dffa,
    scores)
30:            return
31:        else
32:            regionL ← region(r1, a1)
33:            regionR ← region(a2, r2)
34:            good_eps, scores, epsilon ← Golden_section_search(regionL, dffa, scores)
35:            good_eps, scores, epsilon ← Golden_section_search(regionR, dffa, scores)
36:            return
37:        end if
38:    end if
39: end while
```

2.3.5 Merging

After having calculated a good α value, we can finally start the AALERGIA algorithm. RED is a set of states, that is already revised, whereas BLUE is the set of states that we need to look at. The RED set starts with the initial state (line 3) and BLUE gets all its successors (line 4). After that, we loop through all the elements of BLUE and check if we can merge it with the AALERGIA_compatible criterion (line 14). If the states are compatible, we can merge them and the tree becomes smaller, if not, we promote the state to the RED set (line 21) and exclude it from the BLUE set (line 24). In the end, we get a merged DFFA with a good BIC score. Further details on the AALERGIA_compatible criterion can be found at [2]

Algorithm 5 AALERGIA

1: **Input:** a DFFA, a DPFA, the alpha
2: **Output:** the merged DFFA
3: *RED* ← *dffa.initial_state*
4: *BLUE* ← *freq_trans_matrix*^{1}*(dffa.initial_state, all)*
5: *promote* ← *1*
6: **while** BLUE is not empty **do**
7: *BLUE* ← sort(*BLUE*)
8: *q_b* ← *BLUE(1)*
9: *BLUE*.remove*(q_b)*
10: *same_label_ind* ←find_same_labels(*RED, q_b)*
11: **for** *ind* ∈ *same_label_ind* **do**
12: *q_r* ← *RED(ind)*
13: *threshold* ← calculate_compatible_params(*dffa, q_r, q_b, alpha*)
14: **if** AALERGIA_compatible(*dffa, dpfa, q_r, q_b, alpha, threshold*) **then**
15: *dffa_merged* ← AALERGIA_merge(*RED, BLUE, q_r, q_b*)
16: *promote* ← *0*
17: break
18: **end if**
19: **end for**
20: **if** promote **then**
21: *RED* ← *q_b* and *RED*
22: **end if**
23: *successors* ← *dffa_merged.freq_trans_matrix*^{1}(*RED, all*)
24: *BLUE* ← all *successors* not in *RED*
25: **end while**

2.3.6 Perplexity

3 Related Work

[* ME: The contest you were looking at. write three paragraphs each of which dedicated to one of the top three approaches; cite the publications (Worst to best). *] [* ME: The continue the discussion on SHIBATA and why it is winning every major contest in this field PAutomataC 2012, CONTEST 2016. *]

4 Meat

[* ME: a case study on wireless sensory networks. *]

5 Evaluation

[* ME: compare the findings about the case study with real-world scenario and the knowledge of an expert in the field of wireless sensory networks. *]

6 Conclusions

[* ME: the approach is good; but! *]

References

[1] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[2] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, "Learning probabilistic automata for model checking," in *2011 Eighth International Conference on Quantitative Evaluation of Systems (QEST)*, pp. 111–120.

[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.

[4] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic verification of herman's self-stabilisation algorithm," *Formal Aspects of Computing*, vol. 24, no. 4-6, pp. 661–670, 2012.

[5] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Grammatical inference and applications* (R. C. Carrasco, ed.), vol. 862 of *Lecture notes in computer science Lecture notes in artificial intelligence*, pp. 139–152, Berlin: Springer, 1994.

[6] J. Oncina and P. Garcia, "Identifying regular languages in polynomial time," in *Advances in structural and syntactic pattern recognition, Volume 5 of series in machine perception and artificial intelligence*, pp. 99–108, World Scientific, 1992.

[7] C. Shibata and R. Yoshinaka, "The 11th icgi marginalizing out transition probabilities for several subclasses of pfas," 2012.