

Seminarpaper

Mario Wagner, 0730223

Graz, am October 5, 2016

Contents

1	Introduction	1
2	Preliminary	2
2.1	AALERGIA	2
2.1.1	Training Data	2
2.1.2	Frequency Prefix Tree Acceptor	3
2.1.3	Normalization	6
2.1.4	Golden Section Search	6
2.1.5	Merging	10
3	Related Work	11
4	Meat	12
5	Evaluation	13
6	Conclusions	14

List of Figures

1	The Flow Diagram AALERGIA	2
---	-------------------------------------	---

List of Tables

1	The Trainings set	2
2	The Alphabet	3
3	The Prefixes	3
4	The Sorted Prefix Array	5
5	The Transition Matrix	6

6	The Search Region	8
7	The BIC scores	8
8	The Epsilon values	8
9	The Epsilon	8

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).

[* **ME**: here goes a brief history about the L^* algorithm of Angluin (2 or 3 paragraphs). *]

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[1] is an interesting approach to learn stochastic automata from a data-set of traces of a system.

[* **ME**: here you have to reword parts of introduction of AALERGIA [1] that are relate to the incentives of devising such method; two or three paragraphs suffice. *]

2 Preliminary

2.1 AALERGIA

In this paper, the algorithm AALERGIA[1] and the provided MATLAB implementation¹ are used as a starting point for our work. In order to get a better idea of the way the implementation works, we will describe it in detail and a running example for better replicability is used throughout this section. The implementation of the algorithm consists of several parts, which are depicted in the figure 1.



Figure 1: The Flow Diagram AALERGIA

2.1.1 Training Data

The data used to learn the Deterministic Labeled Markov Chain (DLMLC) is supplied by an external *.mat file, which contains MATLAB formatted data: The training set and the alphabet to be used. The alphabet consists of a finite 1xN-cell array, where each column contains one letter in the alphabet. The training set consist of a finite Nx1-cell array, where each row contains sequences of symbols generated from the model, separated by commas.

The running example we are going to use is a self-stabilizing ring network with 3 processes. Table 1 shows the beginning of the trainings set, 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.

000,001,
000,011,100,010,101,010,001,100,010,101,
000,101,
000,111,010,001,110,011,101,010,001,110,011,
000,111,011,101,110,001,
000,101,010,101,110,011,100,011,101,110,

Table 1: The Trainings set

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

¹<http://mi.cs.aau.dk/code/aalergia>

000	001	010	011	100	101	110	111
-----	-----	-----	-----	-----	-----	-----	-----

Table 2: The Alphabet

2.1.2 Frequency Prefix Tree Acceptor

After loading the trainings 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. The frequency shows how often an event in the automaton occurs.

Algorithm 1 shows how the creation of the FPTA is implemented in the AALERGIA package. The code starts by sorting the trainings 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 trainings set and adds them to the cell array named *prefix*, as shown in Table 3.

000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,001,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,
000,000,010,001,100,011,101,110,001,110,001,110,011,101,
000,000,010,001,100,011,101,110,001,110,001,110,011,
000,000,010,001,100,011,101,110,001,110,001,110,
000,000,010,001,100,011,101,110,001,110,001,

Table 3: The Prefixes

Algorithm 1 Create the FPTA

```
1: Input: the  $\Sigma$  and a set of strings  $S$  (training data)
2: Output: a DFFA  $A$  and a sorted set of strings  $U$ 
3:  $U \leftarrow \text{sort}(S)$ 
4:  $prefixes \leftarrow U$ 
5:  $F_{fr} \leftarrow \text{string\_count}(U)$ 
6: for  $u \in U$  do
7:    $index \leftarrow \text{find positions of “,” in } u$ 
8:   while  $index > 0$  do
9:      $substr \leftarrow \text{substring}(1:index)$ 
10:    if  $substr \notin prefix$  then
11:       $prefixes(end+1) \leftarrow substr$ 
12:    end if
13:     $index \leftarrow index-1$ 
14:  end while
15: end for
16:  $prefixes \leftarrow \text{width\_first\_sort}(prefixes)$ 
17:  $predecessor \leftarrow \text{find\_predecessor}(prefixes)$ 
18: for  $p \in prefixes$  do
19:    $sym \leftarrow \text{get\_symbol}(p)$ 
20:    $pre \leftarrow \text{predecessor}(p)$ 
21:    $\text{freq\_trans\_matrix}^1(pre, sym) \leftarrow \text{predecessor}(p)$ 
22:    $\text{frequency\_transition} \leftarrow \text{freq\_trans\_matrix}^2(pre, sym)$ 
23:    $\text{frequency\_transition} \leftarrow \text{frequency\_transition} + \text{frequency}(p)$ 
24:   while  $pre$  do
25:      $\text{update\_freq\_trans\_matrix}(p, pre)$ 
26:      $pre \leftarrow \text{predecessor}(p)$ 
27:   end while
28: end for
29:  $\text{init\_states} \leftarrow \text{freq\_trans\_matrix}^1(1, all)$ 
30: return  $A$ 
```

In our example, the dimension of $prefix$ is 494x1 and contains all unique prefixes of the trainings 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.

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. 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.

000,
000,000,
000,001,
000,010,
000,011,
000,100,
000,101,
000,110,
000,111,
000,000,010,
000,000,100,
000,000,111,
000,001,100,
000,001,110,

Table 4: The Sorted Prefix Array

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)
- BLUE (later used for merging)

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.

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

Table 5: The Transition Matrix

2.1.3 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  $q \in Q$  do
4:    $freq\_state \leftarrow dffa.finalStateFrequency(q)$ 
5:    $ptm^1 \leftarrow dffa.frequencyTransitionMatrix^1$ 
6:    $freq\_trans \leftarrow dffa.frequencyTransitionMatrix^2(q, all)$ 
7:    $freq\_total \leftarrow freq\_trans + freq\_state$ 
8:   if  $freq\_total > 0$  then
9:      $ptm^2(node, index) \leftarrow \frac{freq\_trans}{freq\_total}$ 
10:     $fsp(q) \leftarrow \frac{freq\_state}{freq\_total}$ 
11:   end if
12: end for
13: return  $DPFA(dffa, fsp, ptm)$ 

```

2.1.4 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).

The BIC score 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. A larger BIC score ($x > y$) means a better score and is preferred. Further details are given at [1].

Algorithm 3 Find ε -region

```

1: Input: the DFFA
2: Output: the region for  $\varepsilon$ , scores, epsilon_values
3:  $\varepsilon \leftarrow 1$ 
4:  $score \leftarrow \text{calculate\_BIC\_Score}(\varepsilon, dffa)$ 
5:  $new\_ \varepsilon \leftarrow \varepsilon * 0.5$ 
6: while  $new\_ \varepsilon > 0$  do
7:    $new\_score \leftarrow \text{calculate\_BIC\_score}(new\_ \varepsilon, dffa)$ 
8:   if  $new\_score > score$  then
9:      $region\_right \leftarrow \varepsilon$ 
10:     $\varepsilon \leftarrow new\_ \varepsilon$ 
11:     $new\_ \varepsilon \leftarrow new\_ \varepsilon * 0.5$ 
12:     $score \leftarrow new\_score$ 
13:   else if  $new\_score = score$  then ▷ do this max. 5 times
14:      $\varepsilon \leftarrow new\_ \varepsilon$ 
15:      $new\_ \varepsilon \leftarrow new\_ \varepsilon * 0.5$ 
16:      $score \leftarrow new\_score$ 
17:   else
18:      $region\_left \leftarrow new\_ \varepsilon$ 
19:     break
20:   end if
21: end while
22: ...
23: ...
24: 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 starts with 1 and is halved 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 $-1.5769e + 03$.

0.5	64
-----	----

Table 6: The Search Region

-840.6080	-1.5769e+03	-840.6080	-840.6080	-840.6080	-840.6080	-840.6080	-840.6080
-----------	-------------	-----------	-----------	-----------	-----------	-----------	-----------

Table 7: The BIC scores

1	0.5	2	4	8	16	32	64
---	-----	---	---	---	----	----	----

Table 8: The Epsilon values

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 a_1 and a_2 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 a_1, a_2 for our golden ratio, given the region r_1, r_2 . After that, the corresponding BIC scores f_1, f_2 are calculated (line 9, 10). If the distance between the 2 values a_1, a_2 is small enough, we stop the recursion and the average score between the 2 values is taken as a good ε (line 12). If f_1 is smaller than f_2 , the maximum has to be between a_1 and r_2 and a new golden section search is started. If it is the other way around and f_1 is larger than f_2 , the maximum is between r_1 and a_2 and we start the search with these values. If it happens that the score of f_1 and f_2 is exactly the same, we need to make sure, that the recursion is not endless (line 24). We check if f_1 is larger than the largest score we already calculated. Should this be the case, the maximum is between a_1 and a_2 (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 (r_1, a_1) and the right (a_2, r_2) 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.

64

Table 9: The Epsilon

Algorithm 4 Golden_section_search

```
1: Input: region, dffa, scores
2: Output: good_eps, scores, epsilon
3:  $good\_eps \leftarrow 0$ 
4:  $r1 \leftarrow region(1,1)$ 
5:  $r2 \leftarrow region(1,2)$ 
6: while true do
7:    $a1 \leftarrow r1 + 0.382(r2-r1)$ 
8:    $a2 \leftarrow r2 + 0.618(r2-r1)$ 
9:    $f1 \leftarrow calculate\_BIC\_score(a1, dffa)$ 
10:   $f2 \leftarrow calculate\_BIC\_score(a2, dffa)$ 
11:  if  $|a1-a2| < 0.00001$  then
12:     $good\_eps \leftarrow (a1 + a2) / 2$ 
13:    return
14:  end if
15:  if  $f1 < f2$  then
16:     $region \leftarrow region(a1, r2)$ 
17:     $good\_eps, scores, epsilon \leftarrow Golden\_section\_search(region, dffa, scores)$ 
18:    return
19:  else if  $f1 > f2$  then
20:     $region \leftarrow region(r1, a2)$ 
21:     $good\_eps, scores, epsilon \leftarrow Golden\_section\_search(region, dffa, scores)$ 
22:    return
23:  else
24:    if  $recursion\_depth = 3$  then
25:      return
26:    end if
27:    if  $f1 > max(scores)$  then
28:       $new\_region \leftarrow a1, a2)$ 
29:       $good\_eps, scores, epsilon \leftarrow Golden\_section\_search(new\_region, dffa,$ 
30:  $scores)$ 
31:      return
32:    else
33:       $regionL \leftarrow region(r1, a1)$ 
34:       $regionR \leftarrow region(a2, r2)$ 
35:       $good\_eps, scores, epsilon \leftarrow Golden\_section\_search(regionL, dffa, scores)$ 
36:       $good\_eps, scores, epsilon \leftarrow Golden\_section\_search(regionR, dffa, scores)$ 
37:      return
38:    end if
39:  end while
```

2.1.5 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 [1]

Algorithm 5 AALERGIA

```

1: Input: a DFFA, a DPFA, the alpha
2: Output: the merged DFFA
3:  $RED \leftarrow dffa.initial\_state$ 
4:  $BLUE \leftarrow freq\_trans\_matrix^1(dffa.initial\_state, all)$ 
5:  $promote \leftarrow 1$ 
6: while BLUE is not empty do
7:    $BLUE \leftarrow sort(BLUE)$ 
8:    $q\_b \leftarrow BLUE(1)$ 
9:    $BLUE.remove(q\_b)$ 
10:   $same\_label\_ind \leftarrow find\_same\_labels(RED, q\_b)$ 
11:  for  $ind \in same\_label\_ind$  do
12:     $q\_r \leftarrow RED(ind)$ 
13:     $threshold \leftarrow calculate\_compatible\_params(dffa, q\_r, q\_b, alpha)$ 
14:    if AALERGIA_compatible( $dffa, dpfa, q\_r, q\_b, alpha, threshold$ ) then
15:       $dffa\_merged \leftarrow AALERGIA\_merge(RED, BLUE, q\_r, q\_b)$ 
16:       $promote \leftarrow 0$ 
17:      break
18:    end if
19:  end for
20:  if promote then
21:     $RED \leftarrow q\_b$  and  $RED$ 
22:  end if
23:   $successors \leftarrow dffa\_merged.freq\_trans\_matrix^1(RED, all)$ 
24:   $BLUE \leftarrow all\ successors\ not\ in\ RED$ 
25: end while

```

3 Related Work

Nothing yet.

4 Meat

Nothing yet.

5 Evaluation

Nothing yet.

6 Conclusions

Nothing yet.

References

- [1] 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.