`libalf`: the Automata Learning Framework[*]

(Extended Abstract)

Benedikt Bollig[1], Joost-Pieter Katoen[2], Carsten Kern[2], Martin Leucker[3],
Daniel Neider[2], and David R. Piegdon[2]

[1] LSV, ENS Cachan, CNRS, [2] RWTH Aachen University, [3] TU München

Abstract. This paper presents `libalf`, a comprehensive, open-source library for learning finite-state automata covering various well-known learning techniques (such as, Angluin's L^*, Biermann, and RPNI, as well as a novel learning algorithm for NFA. `libalf` is highly flexible and allows for facilely interchanging learning algorithms and combining domain-specific features in a plug-and-play fashion. Its modular design and its implementation in C++ make it the ideal platform for adding and engineering further, efficient learning algorithms for new target models (e.g., Büchi automata).

1 Introduction

The common objective of all learning algorithms is to generalize knowledge gained throughout a learning process. In such a process, the learning algorithm is confronted with classified examples. They are utilized to derive some kind of hypothesis which is able to classify new examples in conformance with the examples already seen. A conventional way to distinguish learning algorithms is to group them into *online* and *offline* algorithms. Online learning techniques are capable of actively asking queries to some kind of *teacher* who is able to classify these queries. Offline algorithms, on the other hand, are passively provided with a set of classified examples from which they have to build an apposite hypothesis.

In recent years, learning algorithms have become increasingly popular for various application domains and have been successfully used in different fields of computer science, reaching from robotics over pattern recognition (e.g., in bioinformatics) to natural language recognition. Especially in the area of automatic verification, learning techniques have proved their great usefulness. They were used for minimizing partially specified systems [12], model checking blackbox systems [14, 8], and for improving regular model checking [1, 20, 9]. To put it bluntly, automata learning is en vogue.

The need for a unifying framework collecting various types of learning techniques is thus beyond all questions. In addition, it is desirable to have possibilities of easily exchanging or extending the implemented learning algorithms to compare assets and drawbacks for certain user applications. For easing the potential users' lives, a library should provide additional features, such as means for statistical evaluation or loggers. Unfortunately, existing learning frameworks only partly cover these requirements.

The main objective of this paper is to present a new library called the *automata learning framework* (`libalf` for short). `libalf` unifies different kinds of learning

techniques into a single flexible, easy-to-extend, open source library with a clear and easy-to-understand user interface. We would like `libalf` to become a comprehensive compendium of a large variety of learning techniques to which everybody has access in a public domain fashion.

2   Related work

There is a large number of learning algorithms provided in the literature. Usually, the most important and influential ones are implemented again and again by different researchers. Many of these implementations are *quick-and-dirty* applications which are only meant to be a proof-of-concept of the researcher's theoretical work. Most often, this unfortunately implies a lack of extensibility and comparability as the authors did not have time to thoroughly bother for a clear, extensible design. We are only aware of two learning libraries that aim for the objectives mentioned above.

The `LearnLib` library [16, 15] is a package for learning deterministic finite-state automata. It is available as `C++` binary package or via a `CORBA` interface which connects to a dedicated password-protected server located at the University of Dortmund. The `LearnLib` implements Angluin's L^* learning algorithm for inferring DFA and some slight variants for deriving Mealy machines. So-called *filters* can exploit domain-specific properties and, thus, actively reduce the number of queries that a teacher is asked during a learning phase. Moreover, statistical data acquisition can be employed to evaluate the learning procedure. These features make `LearnLib` a powerful tool if a teacher is available and DFA are the target model of choice.

The *Rich Automata Learning and Testing* (`RALT` for short) library [18] has been developed in `Java` yielding a platform independent solution for learning applications. It also implements L^* and three relatives for inferring Mealy machines. In contrast to `LearnLib`, however, it does not make use of any domain-specific optimizations nor are we aware of any external applications using `RALT` as learning library. Regrettably, `RALT` seems not publicly available. In comparison to the `LearnLib` library the comprehensiveness of `RALT` seems to be much lower.

Despite the positive features mentioned above what is clearly missing in both libraries is a way of extending existing learning algorithms or augmenting the libraries with new learning algorithms (also for different kinds of target models). Moreover, there are no implementations for learning nondeterministic automata [4]. As NFA can be exponentially more succinct than DFA they are very interesting for many applications.

3   A library for learning automata: `libalf`

Online algorithms	Offline algorithms
Angluin's L^* [2] (two variants)	Biermann [3]
NL^* [4]	RPNI [13]
Kearns / Vazirani [10]	DeLeTe2 [6]

Table 1. Learning algorithms available in `libalf`.

The `libalf` library is an actively developed, stable, and extensively-tested library for learning finite state machines. It provides a wide range of on- and offline algorithms for learning deterministic (*DFA*) and nondeterministic finite automata (*NFA*): as of today, there are seven algorithms implemented, cf. Table 1.

The main aims of `libalf`'s design are high *flexibility* and *extensibility*. Flexibility, in this context, refers to the user's needs of an easy-to-use but powerful tool, which lets the user concentrate on implementing her ideas without worrying about details of the underlying learning algorithms. `libalf` realizes this twofold: On the one hand, it supports to switch easily between learning algorithms and information sources (often only by changing a single line of code), thus, allowing the user to experiment with different learning techniques. On the other hand, `libalf` fits seamlessly into a huge number of diverse environments as it is available for both C++ and Java (using the Java Native Interface `JNI`) and runs on all familiar operating systems (Windows, Linux and `Mac OS X` in 32- and 64-bit). In addition, the so-called *dispatcher* implements a network-based client-server architecture, which allows one to run `libalf` not only locally but also remotely, e.g., on a high performance machine.

In contrast, the goal of extensibility is to provide easy means to augment the library. This is mainly achieved by `libalf`'s easy-to-extend design and distributing `libalf` freely as open source code.[1]

Finally, `libalf` provides many additional features such as the ability to change the alphabet size during the learning process, extensive logging facilities, domain-based optimizations via so-called *normalizers* and *filters*, GraphViz visualization, etc. A comparison of the most important learning libraries to `libalf` is given in Table 2.

	libalf	LearnLib	RALT
Algorithms	online/offline	online	online
	currently seven	one (L^*)	one (L^*)
Hypotheses	DFA, NFA, Mealy, etc.	DFA, Mealy	DFA, Mealy
Open source	yes	no	n/a
Availability	C++, Java (JNI)	C++	Java
	source code, binary, distribution packages, dispatcher	via Internet connection only (e.g., via CORBA)	n/a
Specifics	filters, normalizers, statistics, visualization	filters, statistics, visualization	visualization

Table 2. Overview over the most important learning libraries in comparison to `libalf`.

Implementation details. `libalf` consists of a C++ library and is complemented by multiple other components: the java interface (`JNI` and dispatcher), `liblangen` (a library to generate random regular languages) and `AMoRE++` (a C++ automata library). All components are available as source-code. The pure `libalf` library consists of approximately 7.300 source lines of code (SLoC). Including all additional components, it comprises 20.000 SLoC.
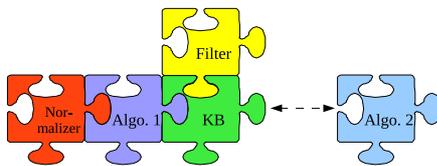

Fig. 1. `libalf`'s modular design allowing for easy composition of learning algorithms, knowledgebases (KB), filters, and normalizers

`libalf`'s main components are the *learning algorithms* and the so-called *knowledgebase*. The *knowledgebase* is an efficient storage for language information and collects queries and classifications thereof. Using an external storage has the advantage of being independent of the choice of the learning algorithm. This makes easy interchange of learning algo-

[1] `libalf` is available on `http://libalf.informatik.rwth-aachen.de/`.

rithms on basis of the same knowledge possible (cf. Figure 1). A knowledgebase can be associated with a number of *filters*, which make use of domain-specific knowledge to reduce the number of queries to the teacher. Such filters can be composed by logical connectors (*and*, *or*, *not*). In contrast, *normalizers* are able to recognize words equivalent in a domain-specific sense to reduce the amount of knowledge that has to be stored. A *logger* is an adjustable logging facility that an algorithm can write to, to ease application debugging and development.

Figure 1 summarizes the modularity of our approach. Different learning algorithms can access one and the same knowledgebase, filters can be associated to knowledgebases, and normalizers and loggers can be added to learning algorithms. Additional learning algorithms can easily be added. To this end, no knowledge about the libraries implementation is required, as well-defined interfaces for all submodules are provided.

`libalf` from a user perspective. The typical workflow of employing a `libalf` learning algorithm is simple. First, we need to provide a teacher (for online algorithms) or a sample set (in an offline setting). Then, we create a knowledgebase, a logger, and an algorithm and associate them. For online-algorithms, we proceed as follows: 1) Let the algorithm advance (i.e., search for a hypothesis or ask membership queries). 2) If no hypothesis is available answer all queries in knowledgebase and goto 1). 3) If a hypothesis is available but wrong, provide a counterexample and goto 1). For offline algorithms the task is easier: Store all available knowledge in knowledgebase and request the algorithm to advance yielding a hypothesis in conformance with the initial input.

This modus operandi does not depend on whether the library is used directly from C++, via `JNI` or via network (dispatcher). The Java interface is even designed to seamlessly switch between `JNI` and dispatcher.

`libalf` in use. Our learning library `libalf` is currently used and extended for inferring CFMs from MSC specifications [5], for learning attractor sets in regular games (D. Neider, RWTH Aachen), and in the setting of probabilistic systems (M. Kwiatkowska, Oxford University). Moreover, there are requests for using `libalf` for searching through source code to find similar code fragments, so-called clones, (E. Jürgen, TU Munich) and for learning black box systems from log files.

Future work. For future work, we plan to augment `libalf` with additional learning algorithms, e.g., [17] or [19]. Another aim is to integrate learning techniques for other important language classes, e.g., transducers [21], Büchi automata [11, 7] etc., and to broaden the scope to the area of probably approximately correct (PAC) learning, e.g. [10]. Another ongoing work puts different learning algorithms in comparison. In this project, we compare different online and offline learning algorithms and evaluate their average time complexity. The results obtained so far look very promising.

4 Conclusion

`libalf` is a new, unifying, broadly-scoped, and open-source learning library, which is easy to use and extend. It gathers several on- and offline learning techniques. The main features of our library and other approaches described previously are summarized in Table 2.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A Survey of Regular Model Checking. In *CONCUR 2004*, volume 3170 of *LNCS*, pages 35–48. Springer, 2004.
2. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. A. W. Biermann and J. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE TSE*, 21(6):592–597, 1972.
4. B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. In *IJCAI 2009*, pages 1004–1009. AAAI Press, 2009.
5. B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. Learning Communicating Automata from MSCs. *IEEE TSE*. To appear.
6. F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using RFSAs. *TCS*, 313(2):267–294, 2004.
7. A. Farzan, Y. Chen, E. M. Clarke, Y. Tsay, and B. Wang. Extending Automated Compositional Verification to the Full Class of Omega-Regular Languages. In *TACAS 2008*, volume 4963 of *LNCS*, pages 2–17. Springer, 2008.
8. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *TACAS*, volume 2280 of *LNCS*, pages 357–370. Springer, 2002.
9. P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. *ENTCS*, 138(3):21–36, 2005.
10. M. Kearns and U. Vazirani. An Introduction to Computational Learning Theory, 1994.
11. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995.
12. A. L. Oliveira and J. P. M. Silva. Efficient Algorithms for the Inference of Minimum Size DFAs. *Machine Learning*, 44(1/2):93–119, 2001.
13. J. Oncina and P. García. Inferring Regular Languages in Polynomial Updated Time. In *the 4th Spanish Symposium on Pattern Recognition and Image Analysis*, volume 1 of *MPAI*, pages 49–61. World Scientific, 1992.
14. D. Peled, M. Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.
15. H. Raffelt and B. Steffen. LearnLib: A Library for Automata Learning and Experimentation. In *FASE 2006*, volume 3922 of *LNCS*, pages 377–380. Springer, 2006.
16. H. Raffelt, B. Steffen, and T. Berg. LearnLib: a library for automata learning and experimentation. In *FMICS*, pages 62–71. ACM, 2005.
17. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–299, 1993.
18. M. Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing*. PhD thesis, Laboratoire Informat. de Grenoble, 2008.
19. B. Trakhtenbrot and J. Barzdin. *Finite automata; behavior and synthesis*. North-Holland, 1973.
20. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to Verify Safety Properties. In *ICFEM 2004*, volume 3308 of *LNCS*, pages 274–289. Springer, 2004.
21. J. Vilar. Query learning of subsequential transducers. In *ICGI*, LNCS, pages 72–83. Springer, 1996.