PAC Learning Automata using Examples

In this project you would have to build a program that implements the probably-approximately correct (PAC) learning model for finite automata.

The entire project is split into sections for ease of comprehension and execution. The sections DO **NOT** indicate an equal distribution of grade, time to be spent, or effort required. Sections 1 and 2 are essential to be completed in a more-than-satisfactory manner for full credit on the project. Section 3 is a bonus section which you may attempt if time permits after everything else is done completely.

You will have to write clean and readable code which compiles. Any code that does not compile will not be graded. You will also provide a short report detailing how each task is handled. This report would also contain a brief but detailed answers for every non-coding task outlined.

Section 0: Introduction and setup

This project involves use of automata. Unless otherwise specified automata (automaton) refer to deterministic finite-state automata (DFAs).

You can use the dk.brics.automaton Java package for handling and manipulating automata. It is well-maintained, has a good documentation and tutorials, and has most features you would need. This is just a suggestion; you can use any programming language and any other package you see fit.

You should be able to input and output automata in a standard human-readable format. The dk.brics.automaton package doesn't have a common read-write to a readable format, so you might have to write a wrapper for the Automaton class.

For simplicity we will consider only binary strings and automata over binary strings.

The modules/functions described are not rigid specifications, and you may slightly vary from function design as long as they have the same functionality and the do the task in the same or in a better way.

Section 1: PAC learning with examples

Within PAC learning for automata using examples, we have a learner who has to infer an automaton. The learner has access to a teacher but no a priori knowledge about the automata. The teacher on the other hand mostly knows the automaton and can provide examples related to the automaton to the learner. An example is a pair of a string and a label which is either + if it is a positive example (i.e. an the automata accepts the string), or - if it is a negative example (i.e. an the automata does not accept the string).
The learner queries for a certain number of examples, and uses these examples to infer the smallest possible automaton which satisfies all the examples.

- Read about **PAC learning of automata from examples**, and get an idea of how learning works and what probably approximately correct means. [**5**%]

You might find the following resources useful:
- Rajesh Parekh, Vasant Honavar. Learning DFA from Simple Examples. *Mach. Learn.* 44, 1-2 (July 2001), 9-35. [Skip the sections on Kolmogrov complexity, and Universal Distribution]
- Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.* 75, 2 (November 1987), 87-106.

Deterministic Finite-state Automaton (DFA) induction is a popular technique to infer a regular language from positive and negative sample strings defined over a finite alphabet. Several state-merging algorithms have been proposed to tackle this task, including RPNI, EDSM etc. These algorithms start from a prefix tree acceptor (PTA), that accepts all the positive examples only, and successively merge states to generalize the induced language.

- Learn what the **RPNI algorithm** and how automata are inferred from examples.[**5**%]

- Create a **learner** module/class which learns an automaton using and examples. This would in effect be like RPNI but instead of having provided the set of examples as input, you would need to get the examples from the teacher module. (How many examples would you need? If this depends on some parameters not mentioned you modify the inputs to the learner module or estimate the parameters reasonably.) [Hint: Read algorithms from the paper mentioned above.] [**20**%]

- Create a **teacher** module/class which answers example queries as mentioned above. This instance of a teacher class will be initiated with the automaton to be learned and will have a function **example** as noted above which the learner can call to generate a single example. For this you may generate examples in a pseudo-random fashion independent of the automata. [**15**%]

The learner and teacher modules should be independent of each other i.e. your learner module should not just work with your teacher module, but any other teacher module implemented by someone else in a completely different way but which has the same specification.

Section 2: Characteristic examples for an automaton

Characteristic set of examples for an automaton is a finite set of (positive and negative) examples that can completely characterize the automaton. A PAC learning module can completely learn the automaton with only examples from the characteristic set.

- Read about **characteristic set** of automata and how they can be created. [**5**%]

- Implement a function **characteristic-set** which takes as input an automaton and outputs a characteristic set of examples for that automaton. [**20**%]

Since there can be several characteristic sets try to obtain the smallest possible one.

- Create a modified teacher module called **characteristic-teacher** such that the examples it provides are from the characteristic set of the automata alone. [**15**%]

- Generate various test cases of varying sizes. While the maximum maximum size of cases your implementation can handle is not specified, try to improve performance as much as possible. See the performance (what parameters would you compare for performance?) of learning based on both the teacher modules. Also try to create unit tests to test each module/function/unit as you go along. [**15**%]

Your code must be sufficiently documented (Javadoc or similar documentation is not necessary but can be included), a build file must be provided, and source, libraries, tests, and results must be cleanly arranged into directories.

Section 3: Bonus Tasks

This section is not essential to the project. This is a **bonus** section, and may count towards extra credit in the course.

- Try to implement PAC learning with examples AND membership queries. [Hint: This is usually done with a classification table or tree. Look up L* learning algorithm] How does this compare to learning with only examples?

The PACS model involves PAC learning with simple examples, i.e. examples drawn according to the conditional Solomonoff-Levin distribution. $P(x) = \lambda_c 2^{-K(x|c)}$, $K(x \mid c)$ denotes the Kolmogorov complexity of x given a representation c of the automata to be learned.

- Try to understand more in detail about this distribution and the PACS model. Can you approximate the **Solomonoff-Levin distribution**, i.e. create a module/class such that given as input an automaton it produces examples as closely as possible to the actual Solomonoff-Levin distribution. This is quite ambitious so don't try this until you have successfully completed all previous tasks.

- Try to add any other features or methods which you might think are interesting or related, or try to optimize any task mentioned.

*** *** ***