An experimental comparison of active learning algorithms on fixed data sets

Du Hyun Cho - 362801 - du.hyun.cho@rwth-aachen.de

March 2025

1 Introduction

The goal of this thesis is to experimentally evaluate performance of active automata learning algorithms when applied in a passive learning setting. In particular, we examine how active learning strategies originally designed for interactive environments with access to an oracle can be adapted to operate on fixed datasets. Learning deterministic finite automata (DFA) from finite samples is a core challenge. In passive learning, learner only has access to a static dataset of labeled words and cannot request additional information. This limitation often makes it difficult to reconstruct the correct structure of the target automaton, especially when data is rare or unrepresentative. By contrast, active learning algorithms such as Angluin's L^* algorithm allow the learner to interact with a teacher (oracle) by posing two types of queries: membership query, which ask whether a given word belongs to the target language, and equivalence query, which ask whether a hypothesized automaton accepts the same language as the target[Angluin, 1987]. If hypothesis is incorrect, teacher returns a counterexample that helps refine the model. In this thesis, we simulate active learning in a passive environment. Learner operates as if it were in an active setting, issuing Membership query and Equivalence query, but responses are derived from fixed datasets. When a membership query refers to a word not present in the dataset, simulated teacher must provide a default answer. Common strategies for this include always answering "yes", always answering "no", answering randomly("yes" or "no") or using nearest-neighbor strategy. However, such guessed answers may be incorrect and lead to flawed generalization, often resulting in an increased number of equivalence queries. This work investigates also how different alternative strategies affect learning outcome. Specifically, we aim to quantify the impact of such strategies on model accuracy, query complexity, and runtime. By conducting experiments on Abbadingo dataset[1]), we aim to analyze trade-offs in accuracy, learning efficiency, and the number of queries required.

The main contributions of this thesis are as follows:

- Design and implementation of a simulated teacher that responds to Membership query and Equivalence query using fixed dataset.
- Evaluation of various strategies for membership queries not found in the dataset.
- Comparison of the learning behavior of algorithms such as L^* and TTT Algorithm across different datasets.

[1] https://abbadingo.cs.nuim.ie/data-sets.html

- Collection and analysis of experimental metrics such as number of membership and equivalence queries, hypothesis accuracy, and runtime.

2 Preliminaries

2.1 Basic Notation

We briefly introduce the basic notation used throughout this thesis. Let Σ be a finite input alphabet. We define the following terms:

- **Words:** A word (or string) over Σ is a finite sequence of symbols from Σ. The set of all such words is denoted by Σ^*. The empty word is denoted by ε.
- **Concatenation:** Given two words $u, v \in \Sigma^*$, their concatenation is written as uv.
- **Prefix and Suffix:** A word u is a prefix of w if there exists $v \in \Sigma^*$ such that $w = uv$. Likewise, v is a suffix of w if $w = uv$ for some $u \in \Sigma^*$.
- **Formal Language:** A language is a subset $L \subseteq \Sigma^*$, representing a collection of accepted words.
- **Deterministic Finite Automaton (DFA):** A DFA is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:
  - Q is a finite set of states,
  - Σ is the input alphabet,
  - $\delta : Q \times \Sigma \to Q$ is the transition function,
  - $q_0 \in Q$ is the initial state,
  - $F \subseteq Q$ is the set of accepting (final) states.
- **Language Accepted by a DFA:** The language accepted by a DFA M is defined as:

$$L(M) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\},$$

  where $\delta(q_0, w)$ is the state reached after processing w from the initial state.

2.2 Active Learning

In this thesis, we use L^* algorithm for automaton learning. It is a well-known active automata learning algorithm introduced by Angluin[Angluin, 1987]. Active automata learning is used to infer behavior of an unknown system, such as constructing a model from an implementation. Learning process relies on two types of queries: membership query and equivalence query. To understand these, we define roles of learner and teacher.

- Learner: A student who tries to figure out the rules of an unknown automaton.
- Teacher: An oracle that knows correct target automaton and can answer queries posed by the learner.

Learner interacts with teacher by asking the following:

- Membership Query : "Given this input word, is it accepted by target automaton?"
- Equivalence Query : "Is the current hypothesis automaton equivalent to the target automaton?"

If hypothesis is incorrect, teacher provides a counterexample — an input word that demonstrates a discrepancy between hypothesis and target. Then what is difference between passive learning and active learning? To illustrate difference between active and passive learning, consider analogy of a vending machine. In an active setting, learner can freely press buttons (i.e., input strings) and immediately observe outcomes (accepted or rejected), updating their understanding accordingly. They can also ask, "Does current hypothesis of "how the machine works" match the real hypothesis?" and receive corrective feedback. In contrast, passive learning offers no such interaction. Learner only has access to a fixed notebook of past button presses and their outcomes. If a new input word is encountered, one not found in the notebook, learner must guess the result based on previous knowledge. This makes learning more challenging and introduces uncertainty into the process. Active learning allows learner to iteratively explore and refine their hypothesis, often leading to faster convergence. On the other hand, Passive learning limits learner to existing data, which reduces efficiency and significantly reduces reliability when data set is sparse or unrepresentative. On the other hand, in active learning, learner can actively ask membership query ("Is this word accepted by the system?") and equivalence query ("Is this hypothesis equivalent to the real system?"). In this thesis, although the setting is fundamentally passive (as we rely on datasets from Abbadingo), we simulate active learning by answering membership and equivalence query based on the available dataset(train.gz and test.gz). This approach exposes key challenges of adapting active learning methods to static datasets, such as risk of incorrect generalization and increased equivalence queries due to uncertainty.

2.3 L* Learning

Angluin's L^* algorithm is a foundational method for active learning of deterministic finite automata (DFAs) through interaction with a teacher[Angluin, 1987]. The L^* algorithm builds an observation table that records how a machine responds to different prefixes and suffixes. It uses a table to form hypotheses about machine's structure and refines hypothesis whenever it finds evidence thatguess is wrong. The main loop is: Ask many Membership queries to fill observation table. Ensure the table is closed and consistent. Build a hypothesis DFA. Ask an Equivalence Query: "Is this a correct machine?" If wrong, receive a counterexample and refine the table. Imagine trying to solve a puzzle where we can only see the pieces one by one. Each membership query is like picking up a piece and placing it on the board. Once we think we have a good guess of the full picture (hypothesis), we ask: "Does this complete puzzle match the real one?" If not, someone gives us a piece that doesn't fit our picture — a counterexample — forcing us to rethink parts of our puzzle. Over time, by carefully organizing what we learn and fixing mistakes, we eventually assemble correct complete image. The observation table acts like a memory book, where we record everything we observe, systematically checking for missing information or inconsistencies. Now we have a look the algorithm L^*.

To understand this algorithm, we need to know what exactly are observation tables, escaping words and closed observation tables. First, an observation table $B = (R, E, f)$ consists of

Algorithm 1 The pseudo-code for L^* is given in Algorithm 1 [Angluin, 1987].

initialization: an observation table $B = (R, E, f)$ with $R{:=}\{\epsilon\}$ and $E{:=}\{\epsilon\}$
repeat
  while (R, E) is not closed or not consistent **do**
    if (R, E) is not closed **then**
      find $r_1 \in R$, $a \in \Sigma^*$ s.t. $row(r_1 a) \neq row(r)$, for all $r \in R$
      $R{:=}R \cup \{r_1 a\}$
    end if
    if (R, E) is not consistent **then**
      find $r_1, r_2 \in R, a \in A$, and $e \in E$ such that $row(r_1) = row(r_1)$ and $L(r_1 ae) \neq L(r_2 ae)$
      $E{:=}E \cup \{ae\}$
    end if
    Make the conjecture $M(R, E)$
    if The teacher answers with no, with a counter-example c **then**
      $R{:=}R \cup prefixes(c)$
    end if
    Until the teacher answers with yes to the conjecture $M(R, E)$.
return $M(R, E)$

- a set $R \subseteq \Sigma^*$ (for the representatives),
- a set $E \subseteq \Sigma^*$ (for the experiments),
- and a function $f\colon (R \cup R \cdot \Sigma) \times E \longrightarrow O$.

To effectively apply the L^* algorithm, it is important to understand the purpose and role of its observation table components. The table consists of three elements: the set of representative prefixes R, the set of experiment suffixes E, and the classification function f. Each of these contributes to the structure and refinement of the learned automaton[Angluin, 1987].

R — Representative Prefixes The set $R \subseteq \Sigma^*$ contains input prefixes that represent paths from the initial state to states within the automaton[Angluin, 1987]. Each prefix in R is used to access a potential state. As such, R serves to explore and differentiate the internal state structure of the unknown DFA.

By ensuring that R includes all necessary prefixes to represent distinguishable states, the learner maintains a sufficiently expressive hypothesis. The table is considered *closed* when no new distinct states are found via extensions of R.

E — Experiment Suffixes The set $E \subseteq \Sigma^*$ consists of suffixes used to probe the behavior of the automaton from the states reached via prefixes in R[Angluin, 1987]. By appending each suffix in E to each prefix in R, the learner observes the system's response and determines whether different prefixes correspond to the same or different states.

These suffixes function as targeted tests that reveal state-specific behavior. If two prefixes in R produce different outputs for the same experiment, they are assumed to lead to distinct states. This ensures the *consistency* of the observation table.

f — Classification Function The function $f : (R \cup R \cdot \Sigma) \times E \to \{0, 1\}$ maps a prefix-suffix pair to an output that indicates whether the concatenated string is accepted by the target language[Angluin, 1987]. The function encapsulates the known behavior of the system and is populated through membership queries or approximated from a static dataset in the passive setting.

The entries of f form the core of the observation table, providing the data from which hypotheses are constructed and refined.

Summary

- R allows the learner to explore the state space by providing access paths.
- E enables the learner to test and distinguish those states.
- f encodes the system's behavior for all relevant combinations.

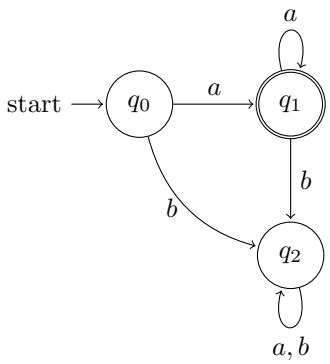
Together, these elements define the structure of the observation table and guide the L^* algorithm in constructing a minimal DFA that accepts the target language[Angluin, 1987].

Example 1

Observation Table

	ϵ	a
ϵ	0	1
a	1	1
ab	0	0
b	0	0
bb	0	0

Automaton for the Observation Table



In the first column of above observation table ϵ, a, ab are the set R and the second and third column ϵ and a are the set E. And we can fill in 0 or 1 by a given automaton. 0 means "reject" and 1 means "accept". If $re \in L$, then $f(r, e) = 1$. Otherwise, $f(r, e) = 0$. For this example, let be $L :=\{w \mid w$ contains only the letter a$\}$. Then choose $r = \epsilon \in R$ and $e = a \in E$. $re = \epsilon a = a \in L$. So $f(r, e) = 1$. And we choose another r and e. If $r = bb$ and $e = a$, then $re = bba \notin L$, because w contains the letter b. In this way, we can interpret the observation table.

Automaton for this observation table is as following: As above automaton, we can find the accepted words: for example, a, aa, aaa, ... The first row and the first column is ϵ. Since ϵ is not accepted by this automaton, there should be 0. And the first row, the second column is ϵa. The state q_0 can reach the state q_1 with the word a. So this word a is accepted by the automaton, therefore there should be 1. In this way, we can make an observation table. Thus, B is an observation table for L if $f(w) = 1 \Leftrightarrow w \in L$.

To correctly infer the structure of an automaton, the observation table must distinguish between different states. Two prefixes should be considered as leading to different states if they behave differently under some suffix. A *reduced* observation table ensures that this is the case: no two rows

are identical unless they truly represent the same state. Intuitively, we can think of each row in the observation table as describing a "fingerprint" of a state — how it reacts to a set of test inputs (E). If two states have the same fingerprint (i.e., same row), we assume they are the same. Thus, a reduced table helps the learner avoid mistakenly merging different states.

Definition 2.1 (reduced observation table)**.** An observation table $B = (R, E, f)$ is called *reduced* if $\forall\ u, v \in R$ with $u \neq v$ there is $w \in E$ with $f(uw) \neq f(vw)$.

Example 2.1(reduced observation table)

	ϵ	a
ϵ	1	0
a	0	1
b	0	0
aa	1	0
ab	0	0
ba	0	0
bb	1	0

Assume the word w is a and $u = a$, $v = b$. Then uw is aa and vw is ba. Have a look at $f(aa)$ and $f(ba)$. $f(aa)$ is 1 and $f(ba)$ is 0. But we have to check for every u and v as well. That means, we should check for $u = \epsilon$, $v = a$ and $u = \epsilon$, $v = b$. $f(\epsilon)$ is 1 and $f(b)$ is 0, so the condition is satisfied. And $f(\epsilon)$ is 1 and $f(a)$ is 0, satisfied as well. Since this condition is satisfied, we can call this table as reduced.

Example 2.2(not reduced observation table)

	ϵ	a
ϵ	1	0
a	0	1
b	0	1
aa	1	0
ab	0	0
ba	0	0
bb	1	0

Here, the rows corresponding to a and b are identical: both are $(0, 1)$ under the suffixes ϵ and a. Therefore, the table is not reduced: it treats a and b as if they reach the same state, even though the actual automaton behaves differently for each.

Now, what if we encounter a new word that doesn't match the fingerprint of any known state? That word is called an *escaping word*. Intuitively, an escaping word reveals that there is a behavior in the target automaton not yet accounted for in the current hypothesis. It behaves differently from all the known prefixes in R when tested with suffixes in E. This signals the presence of a new, unmodeled state, prompting the learner to expand the table and refine the DFA.

Definition 2.2 (escaping word)**.** Assume an observation table B for L is reduced. A word $u \in \Sigma^*$ is called as *escaping* if $\forall\ v \in R$ there is $w \in E$ with $uw \in L \Leftrightarrow vw \notin L$.

Example 3.1

	ϵ	b
ϵ	1	0
a	0	0
b	0	1
aa	1	0
ab	0	0

Assume a given language is $L \subseteq \{a, b\}^*$ with $w \in L$ iff the number of a in w is even and the number of b in w is even. For example, $aa, bb, aabb, bbaa \in L$. Here we choose the word $u = b$. The word v is all representatives ϵ and a, and w is the experiment word in column ϵ or b. Now we check if whether b is escaping or not. Let be first $v = a$ and $w = b$. Then $uw = bb \in L$ but $vw = ab \notin L$. But we have to check one more for case $v = \epsilon$. Then $uw = bb \in L$ but $vw = b \notin L$. (Because the number of a and b should be even.) So the condition for escaping is satisfied, then the word b is escaping. As above L^* algorithm, if a word escaping, we can make a new observation table .

Example 3.2

	ϵ	b
ϵ	1	0
a	0	0
b	0	1
aa	1	0
ab	0	0
ba	0	0
bb	1	0

Since the word b is escaping word, we add this word b in representative column.

Definition 2.3 (Closed table)**.** If there are no words escaping from R, then the table is *closed*.

Example 4

	ϵ	a	b
ϵ	1	0	0
a	0	1	0
b	0	0	1
ab	0	0	0
aa	1	0	0
ba	0	0	0
bb	1	0	0
aba	0	0	1
abb	0	1	0

3 Experimental design and implementation

3.1 Overview

The goal of this experiment is to evaluate how different strategies for answering membership queries in a passive dataset setting influence effectiveness and efficiency of active learning algorithms. Experiment is implemented using Java and LearnLib library[2], and evaluated them using benchmark dataset(e.g., Abbadingo) and randomly generated dataset.

3.2 Active Learning Algorithms

We applied two active learning algorithms:

- L^* – A classic table-based algorithm based on observation tables[Angluin, 1987].
- **TTT** – A more modern tree-based approach to active learning that maintains an evidence tree [Isberner et al., 2014].

Listing 1: Active learning algorithms

```
public class Algorithms {

    // Lstar Setting
    public static ClassicLStarDFA<Character> createLStar(Alphabet<Character> alphabet,
        DFAMembershipOracle<Character> oracle) {
        return new ClassicLStarDFABuilder<Character>()
                .withAlphabet(alphabet)
                .withOracle(oracle)
                .create();
    }

    // TTT Setting
    public static TTTLearnerDFA<Character> createTTT(Alphabet<Character> alphabet,
        DFAMembershipOracle<Character> oracle) {
        return new TTTLearnerDFABuilder<Character>()
                .withAlphabet(alphabet)
                .withOracle(oracle)
                .create();
    }


}
```

3.3 Datasets

3.3.1 Load and Parse

[2]https://learnlib.de/learnlib/maven-site/latest/apidocs/

```java
public class DatasetLoader {

    public static Map<List<Character>, Boolean> load(String filename) throws IOException {
        Map<List<Character>, Boolean> dataset = new LinkedHashMap<>();

        try (BufferedReader reader = getReader(filename)) {
            reader.readLine(); // FIrst line skip (ex: 3478 2)
            String line;

            while ((line = reader.readLine()) != null) {
                String[] tokens = line.trim().split("\\s+");
                boolean label = tokens[0].equals("1");
                int length = Integer.parseInt(tokens[1]);

                List<Character> input = new ArrayList<>();
                for (int i = 0; i < length; i++) {
                    input.add(tokens[2 + i].charAt(0)); // '0' or '1'
                }

                dataset.put(input, label);
            }
        }

        return dataset;
    }

    private static BufferedReader getReader(String filename) throws IOException {
        if (filename.endsWith(".gz")) {
            return new BufferedReader(new InputStreamReader(new GZIPInputStream(new
                FileInputStream(filename))));
        } else {
            return new BufferedReader(new FileReader(filename));
        }
    }
}
```

3.3.2 Abbadingo Dataset

Abbadingo – standardized datasets from the DFA learning competition[3].

```java
Map<List<Character>, Boolean> dataset = DatasetLoader.load("data/train.1.gz");
// Abbadingo Dataset
Map<List<Character>, Boolean> testSet = DatasetLoader.load("data/test.1.gz");
//  Abbadingo testset
```

[3] https://abbadingo.cs.nuim.ie/data-sets.html

Random – randomly generated DFAs with fixed alphabet and state count.

Listing 2: Randomdatasetgenerator.java

```
public class RandomDatasetGenerator {

    public static Map<List<Character>, Boolean> generate(DFA<?, Character> targetDFA,
                                                         Alphabet<Character> alphabet,
                                                         int numSamples,
                                                         int maxLength,
                                                         long seed) {

        Map<List<Character>, Boolean> dataset = new LinkedHashMap<>();
        Random random = new Random(seed);

        for (int i = 0; i < numSamples; i++) {
            int length = random.nextInt(maxLength + 1);
            List<Character> input = new ArrayList<>();
            for (int j = 0; j < length; j++) {
                input.add(alphabet.getSymbol(random.nextInt(alphabet.size())));
            }

            Word<Character> word = Word.fromList(input);
            boolean accepted = targetDFA.accepts(word);

            dataset.put(input, accepted);
        }

        return dataset;
    }
}
```

```
Map<List<Character>, Boolean> dataset = RandomDatasetGenerator.generate(targetDFA,
    alphabet, 100, 6, 42L);
//Random generated dataset
```

3.4 Teacher : Oracle Strategies

To simulate teacher, we implement a class named `ExampleBasedTeacher` which uses abbadingo dataset, randomly generated dataset to answer Membership queries and Equivalence queries. If an Membership query involves a word not in the dataset, the oracle responds using one of the following strategies:

3.4.1 Always Yes

assume the word is accepted.
CODE INSERT

3.4.2 Always No

assume the word is rejected.
CODE INSERT

3.4.3 Random

Just randomly choose an answer between true and false for each round.
CODE INSERT

3.4.4 Nearest Neighbor

return the label of the closest known word using Hamming distance.
CODE INSERT

3.5 Evaluation Metrics

The following metrics are collected for each strategy experiment:

- Number of Membership Queries (MQs)

```
String mqCount = mqOracle.getStatisticalData().getSummary();
System.out.println("MQ:" + mqOracle.getStatisticalData().getSummary());
//print example : "MQ:Queries [#]: 245, "
```

- Number of Equivalence Queries (EQs)

```
DFAExperiment<Character> experiment = new DFAExperiment<>(learner, teacher,
    alphabet);
System.out.println("EQ : " + experiment.getRounds().getCount());
//print example : "EQ : 3"
```

- Size of the learned hypothesis (number of states)

```
DFA<?, Character> hypothesis = experiment.getFinalHypothesis();

System.out.println("States: " + hypothesis.size());
//print example : "States: 13"
```

- Accuracy on testset

```
int correct = 0;
int total = 0;

for (Map.Entry<List<Character>, Boolean> entry : testSet.entrySet()) {
    Word<Character> word = Word.fromList(entry.getKey());
    boolean predicted = dfa.accepts(word);
    boolean actual = entry.getValue();
```

```
        if (predicted == actual) {
            correct++;
        }
        total++;
    }

    double accuracy = total == 0 ? 0.0 : (100.0 * correct / total);

    System.out.println("Accuracy: " + acc + "%");

    //print example : Accuracy: 55.77777777777778%
```

- Runtime in milliseconds

```
    long startTime = System.nanoTime();

    experiment.run();

    long endTime = System.nanoTime();

    long runtimeMillis = (endTime - startTime) / 1_000_000;
    System.out.println("Runtime: " + runtimeMillis + " ms");

    //print example : "Runtime: 240 ms"
```

4 Analyse? Result? +limit?

5 Conclusion

References

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106.

[Isberner et al., 2014] Isberner, M., Howar, F., and Steffen, B. (2014). The ttt algorithm: a redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*, pages 307–322. Springer.