Implementing Kearns-Vazirani Algorithm for Learning DFA Only with Membership Queries

Borja Balle

Laboratori d'Algorísmia Relacional, Complexitat i Aprenentatge
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona
bballe@lsi.upc.edu

Abstract. Two algorithms for learning DFA with membership queries are described. Both of them are based on Kearns and Vazirani's version of Angluin's L^*. Our algorithms tied in the third place in the Zulu competition.

1 Introduction

This short note descibes two algorithms we used to participate in the Zulu competition. This was a competition for algorithms that learn DFA using only membership queries, a problem which has many practical applications (see [3]).

Algorithms entering the competition were given several tasks to solve. Those tasks consisted on learning randomly generated DFA with different number of states n and alphabet sizes $|\Sigma|$. There was a limit in the number of queries a learning algorithm could make for each task. These limits depended on the performance in each particular task of a baseline algorithm provided for the competition: an implementation of Angluin's L^* algorithm [1] using only membership queries. Since the competition's focus was on approximating the target automata, the quality of hypotheses was measured by means of a test set generated from a (mildly) target-dependent distribution. The predicted labels were used to evaluate the hypothesis accuracy.

All notation and definitions used are standard in the literature.

2 Algorithms

In this section we describe two algorithms we entered in the competition — named Balle1 and Balle3 there, L_1 and L_2 here — which tied in the third place. Both algorithms are based on the version of L^* introduced by Kearns and Vazirani in [4], which we will denote by $\mathsf{L}^*_{\mathsf{kv}}$ to avoid confusion. Our two algorithms differ only in the method used to simulate the equivalence queries in $\mathsf{L}^*_{\mathsf{kv}}$ by means of membership queries. The actual implementation was done with Matlab.

Although the differences between L^* and $\mathsf{L}^*_{\mathsf{kv}}$ are subtle enough to be considered implementation issues (cf. [2]), they matter a lot in practical applications. It is known that, given access to membership and equivalence queries, $\mathsf{L}^*_{\mathsf{kv}}$ can be implemented using a number of membership queries that, at least for acyclic DFA, is optimal up to constant factors in the worst case — note this is also the case for the version given by Rivest and Schapire [5]. Furthermore, $\mathsf{L}^*_{\mathsf{kv}}$ presents a feature that is unique among its relatives: the new hypothesis obtained by processing a counterexample s does not necessarily classify s correctly (cf. [4]). This is a valuable property for an algorithm in the Zulu setting, where finding new counterexamples may require spending a large number of membership queries.

Internally, $\mathsf{L}^*_{\mathsf{kv}}$ keeps a data structure called *discrimination tree* that contains information about the states of its current hypothesis, which correspond to equivalence classes of states in the target DFA. Each leaf on the tree corresponds to a different state in the current hypothesis and is identified by a string, the *access string* of that state. A discrimination tree together with a DFA (usually the target to be learned) define a partition of Σ^* in as many sets as leafs in the tree. The process

by which a string s is assigned to a leaf is called *sifting* and goes as follows: starting at the root node, recursively follow the right branch if the DFA accepts the concatenation of s with the string labeling the current internal node of the tree, and follow the left branch otherwise; repeat until a leaf is reached. When constructing a hypothesis from the tree, the transition labeled by σ leaving a state s is directed towards the leaf $\tau(s, \sigma)$ obtained by sifting the word $s\sigma$ down the tree. An example of a discrimination tree plus an hypothesis obtained from it is shown in Fig. 1.

As suggested in [2], our implementation of $\mathsf{L}^*_{\mathsf{kv}}$ uses a caching strategy to avoid repeated queries to the oracle. The procedure for asking membership queries is implemented as a proxy that keeps a dictionary with all the answers already obtained from the oracle. If the algorithm asks a query whose answer is cached in the dictionary, that answer is returned. Otherwise, a new query to the oracle is made and the answer is stored in the dictionary.

Equivalence queries in the original $\mathsf{L}^*_{\mathsf{kv}}$ are replaced in our implementation by a two-layer simulation using membership queries. The first layer is in charge of confronting the current hypothesis with all the answers cached in the dictionary. This takes advantage of the particular feature of $\mathsf{L}^*_{\mathsf{kv}}$ mentioned above. If a counterexample is found in the dictionary, it is returned. Otherwise, the algorithm enters the second layer, detailed in Fig. 2. It is in this second layer where the difference between our two algorithms lies, in particular in the function `Sample`. This function receives as input a length (drawn at random in our case) and outputs a word of that length sampled from a certain distribution — the uniform distribution over Σ^l in the case of L_1. Note that this is the same distribution used in the baseline provided for the Zulu competition.

In the case of L_2 the distribution uses some information obtained from the current hypothesis and discrimination tree, which is based on the following observation: the number of membership queries used for identifying the destination $\tau(s, \sigma)$ of a transition in the hypothesis is the number of steps needed to sift $s\sigma$; that is, the height of the corresponding target leaf in the tree. Based on this observation one can make the heuristic guess that transitions ending in shorter leafs — closer to the root — are less 'informed' than transitions going to taller leafs, and thus more likely to be wrong. Accordingly, in L_2 the distribution used by `Sample` is obtained from a random walk of length l over the hypothesis, with the probability of each transition depending on its destination's height. Under this distribution, strings traversing more transitions towards shorter leafs are more probable. Hopefully, these strings are more likely to be counterexamples to the current hypothesis. In our implementation, a weight is assigned to each transition using the expression

$$w(s, \sigma) = \left(\frac{1}{h_{\tau(s,\sigma)} - h_{\min} + 1} \right)^2 , \tag{1}$$

where $h_{\tau(s,\sigma)}$ is the height of the leaf corresponding to the state $\tau(s, \sigma)$ and $h_{\min}$ is the height of the shortest leaf in the discrimination tree. Transition probabilities are obtained by normalizing these weights for each state: $p(s, \sigma) = w(s, \sigma)/W_s$ where $W_s = \sum_\sigma w(s, \sigma)$. The example hypothesis in Fig. 1 has its transition probabilities computed according to this rule.

3 Discussion

The fact that L_1 performs much better than the baseline — note both of them use the same sampling strategy to search for counterexamples — indicates that implementation issues matter a lot in practical applications.

Even though L_2 seems more principled than L_1, both algorithms performed similarly in the Zulu competition. After the competition, further experiments were conducted on these algorithms, but, surprisingly enough, no statistically significant difference was observed between both algorithms. At the present moment we have no reasonable explanation for this phenomenon.

Acknowledgements. This work is partially supported by: the Generalitat de Catalunya 2009-SGR-1428 (LARCA), the EU PASCAL2 Network of Excellence (FP7-ICT-216886), and an FPU fellowship (AP2008-02064) from the Spanish Ministry of Education.

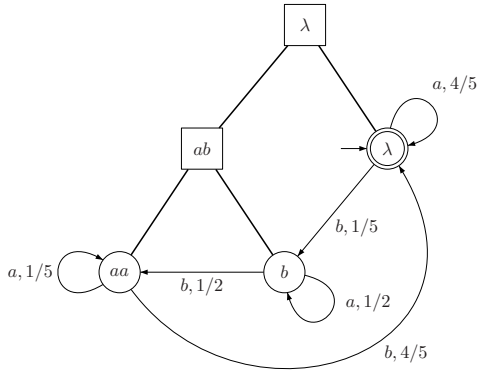

Fig. 1. Discrimination tree with superimposed hypothesis

Input: current hypothesis c and maximum sample size k
Output: a counterexample s or **equal**

for $i \leftarrow 1$ **to** k **do**
  // $U \sim \text{Unif}([0,1])$
  $l \leftarrow \sqrt{U \cdot (\texttt{Depth}(c)+6)^2 + 1}$;
  $s \leftarrow \texttt{Sample}(l)$;
  if $c(s) \neq \texttt{MQ}(s)$ **then return** s;
end
return equal;

Fig. 2. Simulation of an equivalence query with membership queries

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
2. Balcázar, J., Dıaz, J., Gavalda, R.: Algorithms for learning finite automata from queries: A unified view. Advances in Algorithms, Languages, and Complexity pp. 53–72 (1997)
3. Combe, D., Colin, H., Janodet, J.: Zulu: an Interactive Learning Competition (2009)
4. Kearns, M., Vazirani, U.: An introduction to computational learning theory. The MIT Press (1994)
5. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Machine Learning: From Theory to Applications pp. 51–73 (1993)