Interpretable Sequence Classification via Discrete Optimization (Abridged Report)

Maayan Shvo[†], **Andrew C. Li**, **Rodrigo Toro Icarte**, **Sheila A. McIlraith**[†]
University of Toronto & Vector Institute for Artificial Intelligence, Toronto, Canada
[†]Schwartz Reisman Institute for Technology and Society, Toronto, Canada
{maayanshvo, andrewli, rntoro, sheila}@cs.toronto.edu

Abstract

Sequence classification is the task of predicting a class label given a sequence of observations. In many applications such as healthcare monitoring or intrusion detection, early classification is crucial to prompt intervention. In this work, we learn sequence classifiers that favour early classification from an evolving observation trace. While many state-of-the-art sequence classifiers are neural networks, and in particular LSTMs, our classifiers take the form of finite state automata and are learned via discrete optimization. Our automata-based classifiers are interpretable—supporting explanation, counterfactual reasoning, and human-in-the-loop modification—and have strong empirical performance. Experiments over a suite of goal recognition and behaviour classification datasets show our learned automata-based classifiers to have comparable test performance to LSTM-based classifiers, with the added advantage of being interpretable.

The unabridged paper appears here: [28].

1 Introduction

Sequence classification—the task of predicting a class label given a sequence of observations—has a myriad of applications including biological sequence classification (e.g., [6]), document classification (e.g., [26]), and intrusion detection (e.g., [19]). In many settings, early classification is crucial to timely intervention. For example, in hospital neonatal intensive care units, early diagnosis of infants with sepsis (based on the classification of sequence data) can be life-saving [11].

Neural networks such as LSTMs [14], learned via gradient descent, are natural and powerful sequence classifiers (e.g., [34, 16]), but the rationale for classification can be difficult for a human to discern. This is problematic in many domains, where machine decision-making requires a degree of accountability in the form of verifiable guarantees and explanation for decisions [7].

In this work, we use discrete optimization to learn binary classifiers in the form of finite state automata that favour early classification. To classify a sequence of observations, we then employ Bayesian inference to produce a posterior probability distribution over the set of class labels. Importantly, our automata-based classifiers, by virtue of their connection to formal language theory, are both generators and recognizers of the pattern language that describes each behavior or sequence class. We leverage this property in support of a variety of interpretability services, including explanation, counterfactual reasoning, verification, and human modification.

Previous work on learning automata from data has focused on learning minimum-sized automata that perfectly classify the training data (e.g., [10, 1, 23, 32, 2, 9, 29]). Nonetheless, such approaches learn large, overfitted models in noisy domains that generalize poorly to unseen data. We propose novel forms of regularization to improve robustness to noise and introduce an efficient mixed integer linear programming model to learn these automata-based classifiers. Furthermore, to the best of

4th Knowledge Representation and Reasoning Meets Machine Learning Workshop (KR2ML 2020), at NeurIPS.

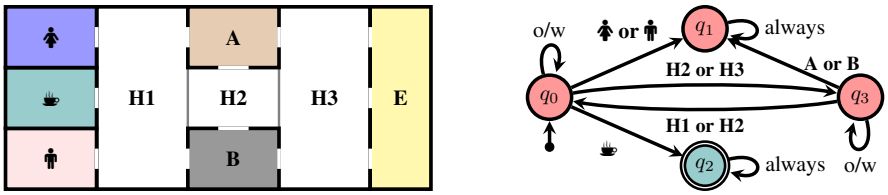

Figure 1: **Left** - Goal recognition environment where the possible goals of the agent are going to an office (**A** or **B**), leaving the building (**E**), going to the restroom (or), or getting coffee ( ). **Right** - a DFA **"get coffee"** classifier, *learned* from data, that detects whether or not the agent is trying to reach the goal . A decision is provided after each new observation based on the current state: **yes** for the blue accepting state, and **no** for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

our knowledge, this is the first work that proposes automata for early classification. Experiments on a collection of synthetic and real-world goal recognition and behaviour classification problems demonstrate that our learned classifiers are robust to noisy sequence data, are well-suited to early prediction, and achieve comparable performance to an LSTM, with the added advantage of being interpretable.

2 Background and Running Example

We consider the problem of classifying noisy sequences of symbolic data where early classification may be favoured. Given a trace $\tau = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, $\sigma_i \in \Sigma$, where Σ is a finite set of symbols, and $\mathcal{C}$ is a set of class labels, sequence classification is the task of predicting the class label $c \in \mathcal{C}$ that corresponds to τ. We propose to use Deterministic Finite Automata as sequence classifiers.

A Deterministic Finite Automaton (DFA) is a tuple $\mathcal{M} = \langle Q, q_0, \Sigma, \delta, F \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite set of symbols, $\delta : Q \times \Sigma \to Q$ is the state-transition function, and $F \subseteq Q$ is a set of accepting states. Given a sequence of input symbols $\tau = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, $\sigma_i \in \Sigma$, a DFA $\mathcal{M} = \langle Q, q_0, \Sigma, \delta, F \rangle$ transitions through the sequence of states $s_0, s_1, \ldots, s_n$ where $s_0 = q_0$, $s_i = \delta(q_{i-1}, \sigma_i)$ for all $1 \leq i \leq n$. $\mathcal{M}$ **accepts** τ if $s_n \in F$, otherwise, $\mathcal{M}$ **rejects** τ.

Example 2.1 (The office domain) *Consider the environment shown in Figure 1. We observe an agent that starts at one of **A**, **B**, or **E** with the goal of reaching one of the other coloured regions, $\mathcal{C} = \{$**A**, **B**, **E**, , , $\}$, using* only *the hallways **H1**, **H2**, and **H3**. The agent always takes the shortest Manhattan distance path to the goal, choosing uniformly at random if multiple shortest paths exist. For example, an agent starting at **B** with goal will pursue paths (**B, H2, H1**) and (**B, H1,**). We wish to predict the agent's goal as early as possible, given a sequence of observed locations.*

Figure 1 depicts a *learned* binary DFA **"get coffee"** classifier. The DFA predicts whether the agent would achieve the goal by keeping track of the agent's locations over time. Its input symbols are $\Sigma = \{$**A**, **B**, **H1**, **H2**, **H3**, **E**, , , $\}$ and the only accepting state is $q_2 \in F$. A decision is provided after each incoming observation based on the current state: **yes** for the blue accepting state, and **no** for red, non-accepting states. For example, on the trace (**B**, **H2**, **H1**,) the DFA would transition through the states (q_0, q_3, q_0, q_2), predicting that the goal is *not* after the first three observations, then predicting the goal *is* after the fourth observation.

3 Learning DFAs for Sequence Classification

Given a set of data traces and corresponding class labels $\{(\tau_1, c_1), \ldots, (\tau_N, c_N)\}$, we train a separate DFA $\mathcal{M}_c$ for each label $c \in \mathcal{C}$ to recognize traces with label c, in a "one-vs-rest" fashion. We start by representing the training set as a *Prefix Tree* (PT) [5], a common preprocessing step in the automata learning literature [9]. Intuitively, a PT is a tree with every training trace as a branch where each node represents a prefix of some training trace. We label each PT node with the number of positive n^+ and negative n^- traces that start with the node's prefix. Positive (resp. negative) refers to traces belonging to (resp. not belonging to) target class c.

We then construct a *Mixed Integer Linear Programming* (MILP) model based on this PT to learn a DFA (see the Appendix § A.2 or [28] for details). The maximum number of DFA states is pre-specified with half the states assigned as accepting states and the other half as rejecting states. The main idea is to assign the unique DFA state q reached by each node n in the PT (representing a sequence of observations) using a binary decision variable x_{nq} (equalling 1 if and only if node n is assigned to DFA state q). If q is an accepting state (resp. rejecting state), the number of *misclassifications* is n^- (resp. n^+) and our model searches for a feasible DFA while minimizing the total number of misclassifications. To reduce overfitting, we penalize the number of (non-self-loop) DFA transitions. Additionally, we designate one accepting state and one non-accepting state of the DFA as *absorbing states* which can only self-transition. These states prevent the classifier from changing decisions once reached. Imposing transition penalties and rewarding nodes for reaching absorbing states are novel regularizers for learning automata and are crucial for generalization.

In order to classify a trace τ when multiple classes are possible, the collective decisions of the DFAs are used to infer a posterior probability distribution over $\mathcal{C}$. A detailed account of our learning approach can be found in the Appendix § A.

4 Classifier Interpretability

An important property of our learned classifiers is that they are interpretable insofar as that the rationale leading to a classification is captured explicitly in the structure of the DFA. DFAs can be queried and manipulated to provide a set of interpretability services including explanation, verification of classifier properties, and (human) modification as elaborated upon in [28]. Here we discuss how explanations may be generated for the classification of some trace and refer the reader to the Appendix for further discussion of the interpretability services afforded by our learned classifiers.

Explanation An important service in support of interpretability is explanation. In the context of classification, given classifier $\mathcal{M}$ and trace τ, we wish to query $\mathcal{M}$, seeking explanation for the classification of τ. In cases where a classifier does not return a positive classification for a trace, a useful explanation can take the form of a so-called *counterfactual explanation* (e.g., [21]).

Let α and β be strings over Σ. The *edit distance* between α and β, $d(\alpha, \beta)$, is equal to the minimum number of edit operations required to transform α to β. Let $\mathcal{M}$ be a DFA classifier that accepts the regular language $\mathcal{L}$ defined over Σ and let τ be some string over Σ. A **counterfactual explanation** for τ is the sequence of edit operations transforming τ to a string $\tau' = \arg\min_{\omega \in \mathcal{L}}(d(\tau, \omega))$. Given the DFA depicted in Figure 1 and a trace $\tau =$ (**A, H2, H1,** 🚹), a possible counterfactual explanation is the edit operation REPLACE 🚹 WITH ☕ which transforms (**A, H2, H1,** 🚹) to (**A, H2, H1,** ☕). This explanation can then be transformed into a natural language sentence: *"The binary classifier would have accepted the trace had* ☕ *been observed instead of* 🚹*"*. A simple approach that generates such natural language sentences from counterfactual explanations can be found in the Appendix § B.1.

5 Experimental Evaluation

In this section we describe the results of an evaluation of our approach, Discrete Optimization for Interpretable Sequence Classification (DISC), on a suite of goal recognition and behaviour classification domains. DISC is the implementation of the MILP model and Bayesian inference method overviewed in Section 3. We compare against LSTM [14], a state-of-the-art neural network architecture for sequence classification; Hidden Markov Model (HMM), a probabilistic generative model which has been extensively applied to sequence tasks [18, 30]; n-gram [8] for $n = 1, 2$, which perform inference under the simplifying assumption that each observation only depends on the last $n - 1$ observations; and a DFA-learning approach (DFA-FT) which maximizes training accuracy (minimizing the number of DFA states only as a secondary objective), representative of existing work in learning DFAs.

Experimental Setup Performance is measured as follows. Cumulative convergence accuracy (CCA) at time t is defined as the percentage of traces τ that are correctly classified given $\min(t, |\tau|)$ observations. Percentage convergence accuracy (PCA) at $X\%$ is defined as the percentage of traces τ that are correctly classified given the first $X\%$ of observations from τ. All results are averaged over 30 runs, unless otherwise specified.

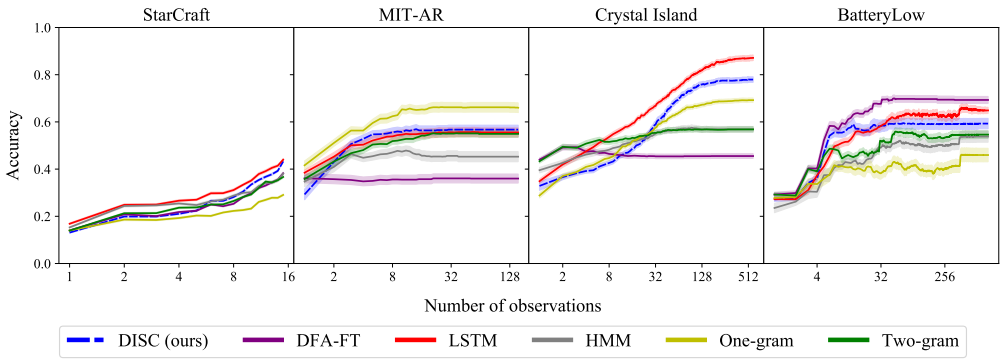

Figure 2: Test accuracy of DISC and all baselines as a function of earliness (number of observations seen so far) on one synthetic dataset (left) and three real-world datasets (three right). We report Cumulative Convergence Accuracy up to the maximum length of a trace. Error bars correspond to a 90% confidence interval. Further results appear in the Appendix § C.

We considered three goal recognition domains: Crystal Island [12], a narrative-based game where players solve a science mystery; ALFRED [27], a virtual-home environment where an agent can interact with various household items and perform a myriad of tasks; and MIT Activity Recognition (MIT-AR) [31], comprised of noisy, real-world sensor data with labelled activities in a home setting. Given a trace the classifier attempts to predict the goal the agent is pursuing.

Experiments for behaviour classification were conducted on a dataset comprising replays of different types of scripted agents in the real-time strategy game StarCraft [15], and on two real-world malware datasets comprising 'actions' taken by different malware applications in response to various Android system events (BootCompleted and BatteryLow) [3]. The behaviour classification task involves predicting the type of StarCraft agent and malware family, respectively, that generated a given behaviour trace.

Results Detailed results for StarCraft, MIT-AR, Crystal Island, and BatteryLow are shown in Figure 2 while a summary of results from all domains (including examples of DFA classifiers learned from the data) is provided in the Appendix and in [28]. DISC generally outperformed n-gram, HMM, and DFA-FT, achieving near-LSTM performance on most domains. LSTM displayed an advantage over DISC on datasets with long traces. n-gram models excelled in some low-data settings (see MIT-AR) but perform poorly overall as they fail to model long-term dependencies. Finally, we conducted an experiment to assess how well DISC performed with respect to "early classification". DISC demonstrated strong performance in each domain - superior to non-LSTM methods and in line with LSTM performance (the results are presented in the Appendix § C.4).

6 Concluding Remarks

The classification of (noisy) symbolic time-series data represents a significant class of real-world problems that includes malware detection, transaction auditing, fraud detection, and a diversity of goal and behavior recognition tasks. The ability to interpret and troubleshoot these models is critical in most real-world settings. In this paper we proposed a method to address this class of problems by combining the learning of DFA sequence classifiers via MILP with Bayesian inference. Our approach introduced novel automata-learning techniques crucial to addressing regularization, efficiency, and early classification. Critically, the resulting DFA classifiers offer a set of interpretability services that include explanation, counterfactual reasoning, verification of properties, and human modification. Our implemented system, DISC, achieves similar performance to LSTMs and superior performance to HMMs and n-grams on a set of synthetic and real-world datasets, with the important advantage of being interpretable.

Acknowledgments and Disclosure of Funding

We gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada CIFAR AI Chairs Program, and Microsoft Research. The third author also acknowledges funding from ANID (Becas Chile). Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute for Artificial Intelligence (`www.vectorinstitute.ai/partners`). Finally, we thank the Schwartz Reisman Institute for Technology and Society for providing a rich multi-disciplinary research environment.

References

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

[2] D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[3] M. L. Bernardi, M. Cimitile, D. Distante, F. Martinelli, and F. Mercaldo. Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security*, 18(3):257–284, 2019.

[4] A. Camacho and S. A. McIlraith. Learning interpretable models expressed in linear temporal logic. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 621–630, 2019.

[5] C. De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[6] M. Deshpande and G. Karypis. Evaluation of techniques for classifying biological sequences. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 417–431. Springer, 2002.

[7] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[8] T. Dunning. *Statistical identification of language*. Computing Research Laboratory, New Mexico State University Las Cruces, NM, USA, 1994.

[9] G. Giantamidis and S. Tripakis. Learning moore machines from input-output traces. In *International Symposium on Formal Methods*, pages 291–309. Springer, 2016.

[10] E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[11] M. P. Griffin and J. R. Moorman. Toward the early diagnosis of neonatal sepsis and sepsis-like illness using novel heart rate analysis. *Pediatrics*, 107(1):97–104, 2001.

[12] E. Y. Ha, J. P. Rowe, B. W. Mott, and J. C. Lester. Goal recognition with markov logic networks for player-adaptive games. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[13] H. Harman and P. Simoens. Action graphs for proactive robot assistance in smart environments. *JOURNAL OF AMBIENT INTELLIGENCE AND SMART ENVIRONMENTS*, 12(2):79–99, 2020.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[15] P. Kantharaju, S. Ontañón, and C. W. Geib. Scaling up CCG-Based Plan Recognition via Monte-Carlo Tree Search. In *Proc. of the IEEE Conference on Games 2019*, 2019.

[16] F. Karim, S. Majumdar, H. Darabi, and S. Harford. Multivariate lstm-fcns for time series classification. *Neural Networks*, 116:237–245, 2019.

[17] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] J. Kupiec. Robust part-of-speech tagging using a hidden markov model. *Computer speech & language*, 6(3):225–242, 1992.

[19] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):295–331, 1999.

[20] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL-the planning domain definition language, 1998.

[21] T. Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019.

[22] W. Min, B. W. Mott, J. P. Rowe, B. Liu, and J. C. Lester. Player goal recognition in open-world digital games with long short-term memory networks. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2590–2596, 2016.

[23] J. Oncina and P. Garcia. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108. World Scientific, 1992.

[24] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[25] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*. Springer Science & Business Media, 2012.

[26] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[27] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. URL `https://arxiv.org/abs/1912.01734`.

[28] M. Shvo, A. C. Li, R. Toro Icarte, and S. A. McIlraith. Interpretable Sequence Classification via Discrete Optimization. *arXiv preprint arXiv:2010.02819*, 2020.

[29] R. Smetsers, P. Fiterău-Broştean, and F. Vaandrager. Model learning as a satisfiability modulo theories problem. In *International Conference on Language and Automata Theory and Applications*, pages 182–194. Springer, 2018.

[30] E. L. Sonnhammer, G. Von Heijne, A. Krogh, et al. A hidden markov model for predicting transmembrane helices in protein sequences. In *Ismb*, volume 6, pages 175–182, 1998.

[31] E. M. Tapia, S. S. Intille, and K. Larson. Activity recognition in the home using simple and ubiquitous sensors. In *International conference on pervasive computing*, pages 158–175. Springer, 2004.

[32] V. Ulyantsev, I. Zakirzyanov, and A. Shalyto. Bfs-based symmetry breaking predicates for dfa identification. In *International Conference on Language and Automata Theory and Applications*, pages 611–622. Springer, 2015.

[33] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[34] C. Zhou, C. Sun, Z. Liu, and F. Lau. A c-lstm neural network for text classification. *arXiv preprint arXiv:1511.08630*, 2015.

Appendix

Table of Contents

In Appendix A, we provide further details concering our procedure to learn a DFA-based classifier from a training set. In Appendix B, we outline a simple natural language generation approach for counterfactual explanation, present samples of learned DFA classifiers from our experiments, discuss classifier verification and modification, and provide exposition of Linear Temporal Logic. In Appendix C, we provide additional details of our experimental setup and our datasets and present additional experimental results (including results from early and multi-label classification experiments).

A Learning DFAs from Training Data

In this appendix, we provide further details concerning our procedure to learn a DFA-based classifier from a training set $\mathcal{T} = \{(\tau_1, c_1), \ldots, (\tau_N, c_N)\}$. Recall that, for each possible label $c \in \mathcal{C}$, we train a separate DFA, $\mathcal{M}_c$, responsible for recognizing traces with label c. We then use those DFAs to compute a probability distribution for online classification of partial traces.

A.1 From Training Data to Prefix Trees

The first step to learning a DFA is to construct a Prefix Tree (PT). Algorithm 1 shows the pseudo-code to do so. It receives the training set $\mathcal{T}$ and the label c^+ of the positive class. It returns the PT for that training set and class label. It also labels each PT node with the costs associated with classifying that node as positive and negative, respectively. That cost depends on the length of the trace and its label, and it is computed using the function `add_cost`().

A.2 From Prefix Trees to DFAs

We now discuss the MILP model we use to learn a DFA given a PT. The complete MILP model follows.

$$\min \sum_{n \in N} c_n + \lambda_e \sum_{q \in Q} \sum_{\sigma \in \Sigma} e_{q,\sigma} + \lambda_t \sum_{n \in N} t_n \qquad \texttt{(MILP)}$$

Algorithm 1 Converting Training Data into Prefix Trees

```
function GET_PREFIX_TREE(𝒯, c⁺)
    r ← create_root_node()
    for (τ, c) ∈ 𝒯 do
        n ← r
        add_cost(n, c = c⁺, |τ|)
        for σ ∈ τ do
            if not has_child(n,σ) then
                add_child(n,σ)
            n ← get_child(n,σ)
            add_cost(n, c = c⁺, |τ|)
    return r
```

$$
\begin{aligned}
s.t.\ & \sum_{q \in Q} x_{n,q} = 1 && \forall n \in N && (1)\\
& x_{r,0} = 1 && && (2)\\
& \sum_{q' \in Q} \delta_{q,\sigma,q'} = 1 && \forall q \in Q, \sigma \in \Sigma && (3)\\
& \delta_{q,\sigma,q} = 1 && \forall q \in T, \sigma \in \Sigma && (4)\\
& x_{p(n),q} + x_{n,q'} - 1 \leq \delta_{q,s(n),q'} && \forall n \in N \setminus \{r\}, q \in Q, q' \in Q && (5)\\
& c_n = \lambda^+ \sum_{q \in F} c^+(n) x_{n,q} + \lambda^- \sum_{q \in Q \setminus F} c^-(n) x_{n,q} && \forall n \in N && (6)\\
& e_{q,\sigma} = \sum_{q' \in Q \setminus \{q\}} \delta_{q,\sigma,q'} && \forall q \in Q, \sigma \in \Sigma && (7)\\
& t_n = \sum_{q \in Q \setminus T} x_{n,q} && \forall n \in N && (8)\\
& x_{n,q} \in \{0,1\} && \forall n \in N, q \in Q && (9)\\
& \delta_{q,\sigma,q'} \in \{0,1\} && \forall q \in Q, \sigma \in \Sigma, q' \in Q && (10)\\
& c_n \in \mathbb{R} && \forall n \in N && (11)\\
& e_{q,\sigma} \in \mathbb{R} && \forall q \in Q, \sigma \in \Sigma && (12)\\
& t_n \in \mathbb{R} && \forall n \in N && (13)
\end{aligned}
$$

This model learns a DFA over a vocabulary Σ with at most $q_{\max}$ states. From those potential states Q, we set state 0 to be the initial state q_0 and predefine a set of accepting states $F \subset Q$ and a set of terminal states $T \subset Q$. We also use the following notation to refer to nodes in the PT: r is the root node, $p(n)$ is the parent of node n, $s(n)$ is the symbol that caused node $p(n)$ to transition to node n, $c^+(n)$ is the cost associated with predicting node n as positive, $c^-(n)$ is the cost associated with predicting node n as negative, and N is the set of all PT nodes. The model also has hyperparameters λ_e and λ_t to weight our regularizers and hyperparameters λ^+ and λ^- to penalize misclassifications of positive and negative examples differently (in the case where the training data is imbalanced).

The idea behind our model is to assign DFA states to each node in the tree. Then, we look for an assignment that is feasible (i.e., it can be produced by a deterministic DFA) and optimizes a particular objective function—which we describe later. The main decision variables are $x_{n,q}$ and $\delta_{q,\sigma,q'}$, both binary. Variable $x_{n,q}$ is 1 iff node $n \in N$ is assigned the DFA state $q \in Q$. Variable $\delta_{q,\sigma,q'}$ is 1 iff the DFA transitions from state $q \in Q$ to state $q' \in Q$ given symbol $\sigma \in \Sigma$. Note that c_n, $e_{q,\sigma}$, and t_n are auxiliary (continuous) variables used to compute the cost of the DFAs.

Constraint (1) ensures that only one DFA state is assigned to every PT node and constraint (2) forces the root node to be assigned to q_0. Constraint (3) ensures that the DFA is deterministic and constraint (4) makes the terminal nodes sink nodes. Finally, constraint (5) ensures that the assignment can be emulated by the DFA. The rest of the constraints compute the cost of solutions and the domain of the variables. In particular, note that the objective function minimizes the prediction error using c_n, the

number of transitions between different DFA states using $e_{q,\sigma}$, and the occupancy of non-terminal states using t_n.

This model has $O(|N||Q| + |\Sigma||Q|^2)$ decision variables and $O(|N||Q|^2 + |\Sigma||Q|)$ constraints.

A.3 Derivation of the Posterior Probability Distribution over the Set of Class Labels

Given a trace (or partial trace) τ and the decisions of the one-vs-rest classifiers $\{\mathrm{D}_c(\tau) : c \in \mathcal{C}\}$, we use an approximate Bayesian method to infer a posterior probability distribution over the true label c^*. Each $D_c(\tau)$ is treated as a discrete random variable with possible outcomes $\{\mathrm{accept}, \mathrm{reject}\}$. We make the following assumptions: (1) the classification decisions D_c for $c \in \mathcal{C}$ are conditionally independent given the true label c^* and (2) $p(D_c|c^*)$ only depends on whether $c = c^*$.

For each c', we compute the posterior probability of $c^* = c'$ to be

$$
\begin{aligned}
& p(c^* = c' | \{\mathrm{D}_c : c \in \mathcal{C}\}) \\
\propto\ & p(c^* = c') * p(\{\mathrm{D}_c : c \in \mathcal{C}\} | c^* = c') \quad \text{(using Bayes' rule)} \\
=\ & p(c^* = c') * \prod_{c \in \mathcal{C}} p(D_c | c^* = c') \quad \text{(using (1))} \\
=\ & p(c^* = c') * p(D_{c'} | c^* = c') * \prod_{c \in \mathcal{C} \setminus \{c'\}} p(D_c | c^* \neq c) \quad \text{(using (2))} \\
\propto\ & \frac{p(c^* = c') * p(D_{c'} | c^* = c')}{p(D_{c'} | c^* \neq c')} \quad \text{(dividing through by the constant } \prod_{c \in \mathcal{C}} p(D_c | c^* \neq c)\text{)}
\end{aligned}
$$

The probabilities on the right-hand side are estimated using a held-out validation set. In particular, the prior probability $p(c^* = c')$ is the approximate proportion of examples with label c', $p(D_{c'} = \mathrm{accept} | c^* = c')$ is the recall of the classifier for label c', and so on. We normalize the posterior probabilities $p(c^* = c' | \{\mathrm{D}_c : c \in \mathcal{C}\})$ such that their sum over $c' \in \mathcal{C}$ is 1 to obtain a valid probability distribution. Note that we could potentially improve the inference by further conditioning on the number of observations seen so far, or relaxing assumption (2). However, this would substantially increase the number of probabilities to be estimated and result in less accurate estimates in low-data settings.

B Interpretability

DFAs provide a compact, graphical representation of a (potentially infinite) set of traces the DFA positively classifies. While many people will find the DFA structure highly interpretable, the DFA classifier can be transformed into a variety of different language-preserving representations including regular expressions, context-free grammars (CFGs), and variants of Linear Temporal Logic (LTL) [24] (see also Appendix § B.5). These transformations are automatic and can be decorated with natural language to further enhance human interpretation (see also Appendix § B.4).

B.1 Natural Language Generation for Counterfactual Explanation

In Section 4 of our paper, we discussed counterfactual explanations, which are useful in cases where a classifier does not return a positive classification for a trace. Here we describe a simple algorithm that transforms a counterfactual explanation (comprising a sequence of edit operations) to an English sentence. We define three edit operations over strings: REPLACE(s, c_1, c_2) replaces the first occurrence of the character c_1 in the string s with the character c_2; INSERT(s, c_1, c_2) inserts the character c_1 after the first occurrence of the character c_2 in the string s; DELETE(s, c_1) removes the first occurrence of the character c_1 from the string s.

Algorithm 2 accepts as input a sequence of edit operations $e_1, e_2, \ldots, e_n$ (where e_i is either a replace, insert, or delete operation), and returns a string representing an English sentence encoding the counterfactual explanation for some trace τ. Redundant “and"s are removed from the resulting string. We do not consider multiple occurrences of the same character in a single string but this can be easily handled. $e_i.args[i]$ is assumed to return the $i+1$th argument of an edit operation e_i and $e_i.type$ is assumed to return the type of the edit operation (e.g., REPLACE). CONCATENATE(s_1, s_2) appends

the string s_2 to the suffix of the string s_1. We further assume that connectives (e.g., 'and') are added between the substrings representing the edit operations.

Algorithm 2 Natural Language Generation for Counterfactual Explanation

Require: A sequence of edit operations $E = e_1, e_2, \ldots, e_n$
1: $s \leftarrow$ *"The binary classifier would have accepted the trace"*
2: For e in E:
3: If e.type == REPLACE
4: CONCATENATE(s, *" had* $e.args[2]$ *been observed instead of* $e.args[1]$*"*)
5: If e_i.type == INSERT
6: CONCATENATE(s, *" had* $e.args[2]$ *been observed following the observation of* $e.args[1]$*"*)
7: If e.type == DELETE
8: CONCATENATE(s, *" had* $e.args[1]$ *been removed from the trace"*)
9: RETURN s

For example, using the DFA depicted in Figure 1, if $\tau =$ (**A, H2, H1,** 🧍) then a possible counterfactual explanation is the edit operation REPLACE(τ, 🧍, ☕) which transforms (**A, H2, H1,** 🧍) to (**A, H2, H1,** ☕). Given the edit operation REPLACE(τ, 🧍, ☕), Algorithm 2 returns the string *"The binary classifier would have accepted the trace had* ☕ *been observed instead of* 🧍*"*.

B.2 Samples of Learned DFA Classifiers

In this appendix we present a number of examples of DFAs learned by DISC from our experimental evaluation in Section 5 of our paper. As discussed in Section 4, our purpose in this work is to highlight the breadth of interpretability services afforded by DFA classifiers via their relationship to formal language theory. The effectiveness of a particular interpretability service is user-, domain-, and even task-specific and is best evaluated in the context of individual domains. Moreover, the DFAs presented in this appendix require familiarity with the domain in question and are therefore best suited for domain experts.

B.2.1 Malware

We present two DFAs learned via DISC from the real-world *malware* datasets. Figure 3 depicts a DFA classifier for BatteryLow that detects whether a trace of Android system calls was issued by the malware family *DroidKungFu4*. The maximum number of states is limited to 10. The trace $\tau =$ (*sendto, epoll_wait, recvfrom, gettid, getpid, read*) is rejected by the depicted *DroidKungFu4* DFA classifier. This can be seen by starting at the initial state of the DFA, q_0, and mentally following the DFA transitions corresponding to the symbols in the trace. The exercise of following the symbols of the trace transition through the DFA can be done by anyone. For the domain expert, the symbols have meaning (and can be replaced by natural language words that are even more evocative, as necessary). In this trace we see that rather than stopping at accepting state q_6, the trace transitions in the DFA to q_5, a non-accepting state. One counterfactual explanation that our system generates to address what changes could result in a positive classification is: "The binary classifier would have accepted the trace had *read* been removed from the trace" (per the algorithm in Appendix B.1).

For comparison, Figure 4 presents a smaller DFA for BootCompleted, learned with the maximum number of states limited to 5. (This DFA was not used in our experiments.) The DFA detects whether a trace was issued by the malware family *DroidDream*. The trace (here truncated, as subsequent observations do not affect the classification decision) $\tau =$ (*clock_gettime, epoll_wait, clock_gettime, clock_gettime, getpid, writev, ...*) is rejected by the DFA. One counterfactual explanation that our system generates to address what changes could result in a positive classification is: "The binary classifier would have accepted the trace had *getuid32* been observed instead of *writev*".

Note that while the 5-state DFA may be more interpretable to humans than the 10-state DFA, the 10-state DFA can model more complex patterns in the data. Indeed, during the course of our experiments with the malware datasets, we found that setting $q_{\max} = 10$ achieved superior performance to $q_{\max} = 5$.

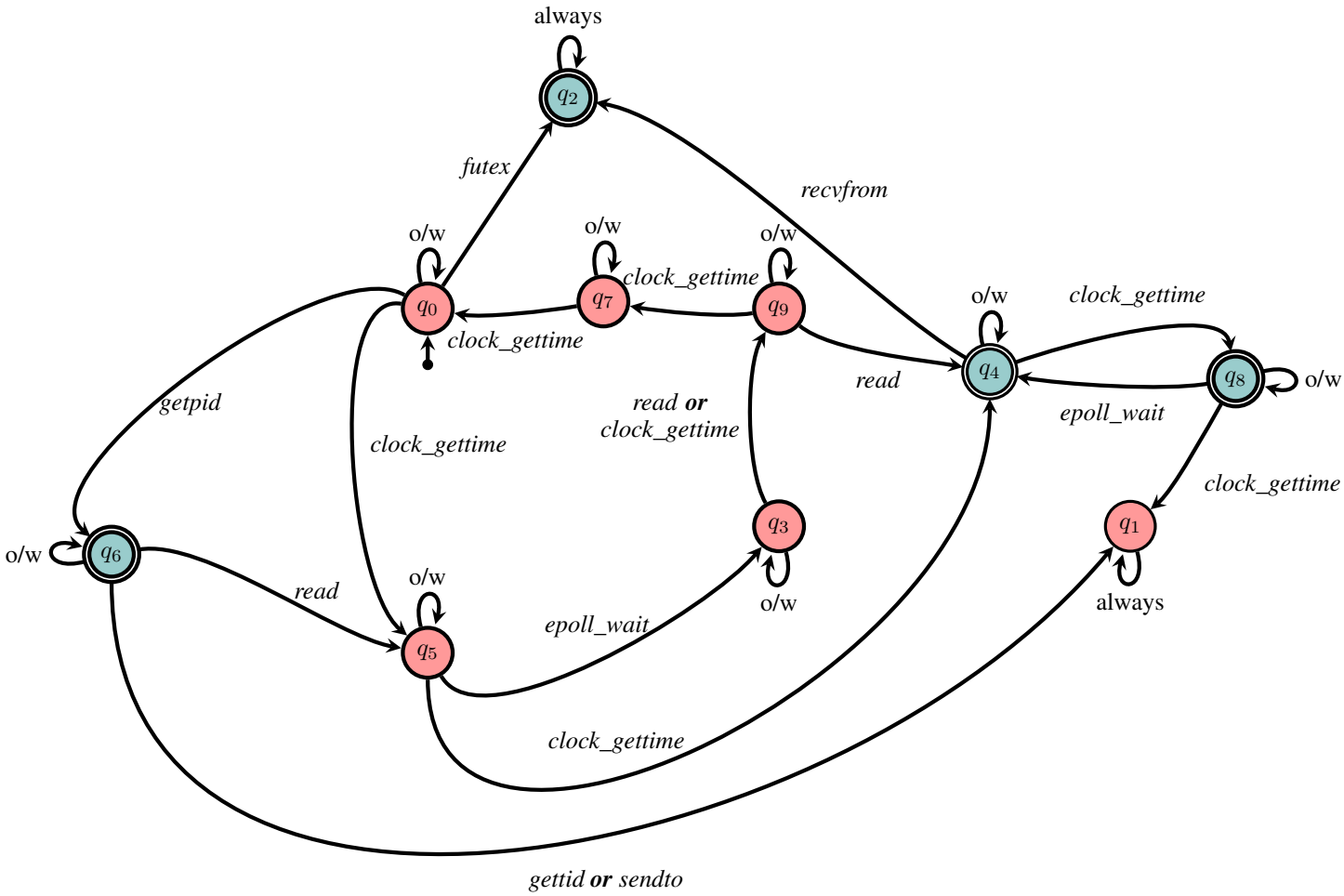

Figure 3: A DFA learned in our experiments from the BatteryLow dataset by limiting the maximum number of states to 10. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. “o/w” (otherwise) stands for all symbols that do not appear on outgoing edges from a state. “always” stands for all symbols.

B.2.2 Crystal Island

Figure 5 depicts a DFA classifier learned from the Crystal Island dataset that detects whether a trace of player actions was performed in order to achieve the goal *Talked-to-Ford*. The maximum number of states is limited to 5.

Consider the trace $\tau =$ (*pickup banana, move outdoors (2a), move outdoors (2b), open door infirmary bathroom, move outdoors (3a), move hall*), which is rejected by the depicted *Talked-to-Ford* DFA classifier. One possible counterfactual explanation to result in a positive trace is: “The binary classifier would have accepted the trace had *move sittingarea* been observed following the observation of *move hall*". Additionally, a *necessary condition* for this DFA to accept is that either *talk ford* or *move sittingarea* is observed — or equivalently, the LTL property $\Diamond$ (*talk ford* $\vee$ *move sittingarea*). This LTL formula is entailed by the DFA.

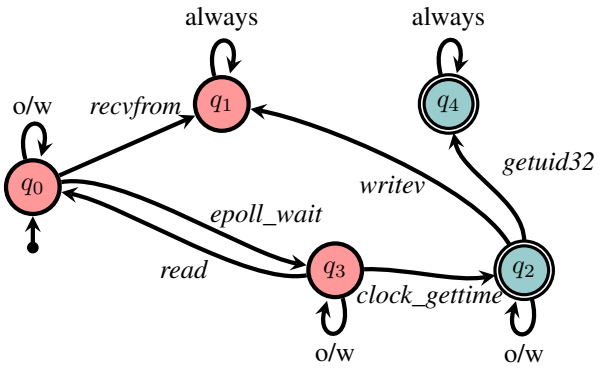

Figure 4: A DFA learned from the BootCompleted dataset by limiting the maximum number of states to 5. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

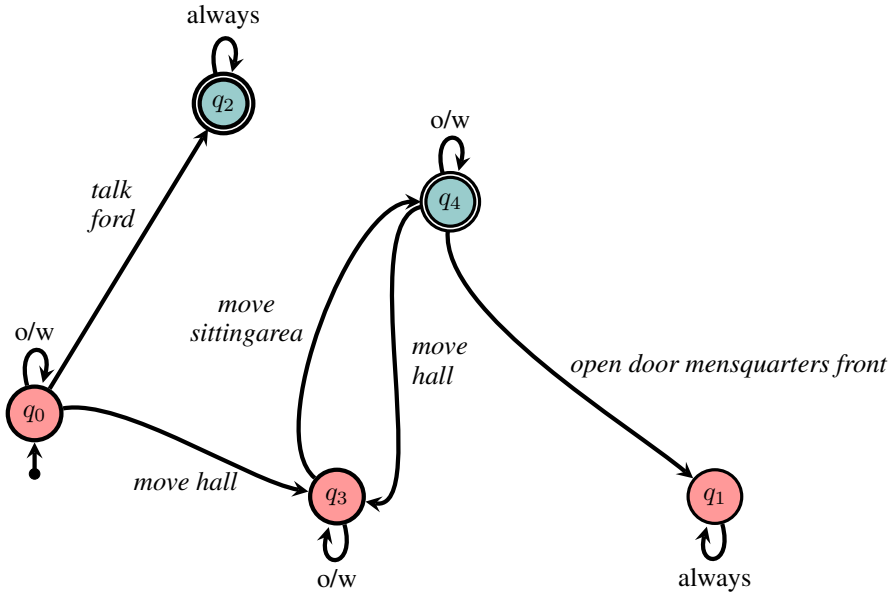

Figure 5: A DFA learned from the Crystal Island dataset, limiting the maximum number of states to 5. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

B.2.3 StarCraft

Figure 6 depicts a DFA classifier learned from the StarCraft dataset that detects whether a trace of actions was generated by the StarCraft-playing agent *EconomyMilitaryRush*. The maximum number of states is limited to 10.

The trace $\tau =$ (*move produce, produce, move produce, move, move, move, harvest, harvest, harvest, move produce, move produce, attack move move*) is rejected by the depicted *EconomyMilitaryRush* DFA classifier. A counterfactual explanation to result in a positive trace is: "The binary classifier would have accepted the trace had *harvest move* been observed instead of *attack move move*". A *necessary condition* for the DFA to accept is that either *move produce* or *harvest produce* is observed in the trace. Furthermore, every trace starting with *harvest produce* will be accepted. As discussed in Section 4, these properties can be automatically extracted from the DFAs.

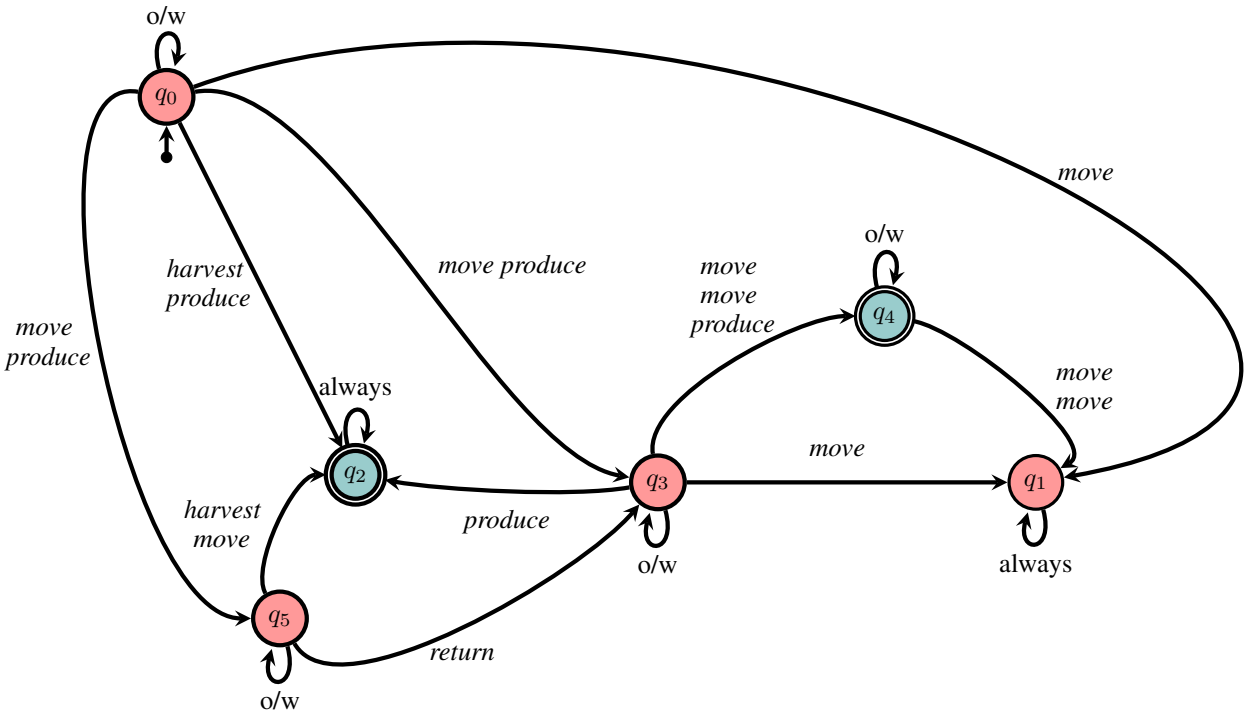

Figure 6: A DFA learned from the StarCraft dataset by limiting the maximum number of states to 10. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

B.3 Transformation of Learned DFA Classifiers to Language-preserving Representations

As mentioned in Section 4, DFAs provide a compact, graphical representation of a (potentially infinite) set of traces the DFA positively classifies. While many people will find the DFA structure highly interpretable, the DFA classifier can be transformed into a variety of different language-preserving representations including regular expressions, context-free grammars (CFGs), and variants of Linear Temporal Logic (LTL) [24]. These transformations are automatic and can be decorated with natural language to further enhance human interpretation.

Example B.1 *The following regular expression compactly describes the set of traces that are classified as belonging to the DFA classifier depicted in Figure 1:* $[(\Sigma - \{🚺, 🚹, H2, H3\})^*(H2 \cup H3)(\Sigma - \{A, B, H1, H2\})^*(H1 \cup H2)]^*(\Sigma - \{🚺, 🚹, H2, H3\})^*☕\Sigma^*$

Of course, this regular expression is only decipherable to a subset of computer scientists. We include it in order to illustrate/demonstrate the multiple avenues for interpretation afforded by our DFA classifiers. In particular, the regular expression can be further transformed into a more human-readable form as illustrated in Example B.2 or transformed into a CFG that is augmented with natural language in order to provide an enumeration, or if abstracted, a compact description of the traces accepted by the DFA classifier.

Example B.2 *The regular expression can be transformed into a more readable form such as: "Without first doing 🚺 or 🚹, repeat the following zero or more times: eventually do H2 or H3, then without doing A or B, eventually do H1 or H2, followed optionally by other events, excluding 🚺 and 🚹. Finally do ☕, followed by anything."*

For others, it may be informative to extract path properties of a DFA as LTL formulae, perhaps over a subset of Σ or with preference for particular syntactic structures (e.g., [4]).

Example B.3 *DFA classifier $\mathcal{M} \models \Box\Diamond$ ☕, the LTL property "always eventually do ☕ ".*

These transformations and entailments utilize well studied techniques from formal language theory (e.g., [25]). Which delivery form is most effective is generally user- and/or task-specific and should be evaluated in situ via a usability study.

B.4 Classifier Verification and Modification

Explanation encourages human trust in a classification system, but it can also expose rationale that prompts a human (or automated system) to further question or to wish to modify the classifier. Temporal properties of the DFA classifier $\mathcal{M}$, such as *"Neither 🚺 nor 🚹 occur before ☕"* can be straightforwardly specified in LTL and verified against $\mathcal{M}$ using standard formal methods verification techniques (e.g., [33]). In the case where the property is false, a witness can be be returned.

Our learned classifiers are also amenable to the inclusion of additional classification criteria, and the modification to the DFA classifier can be realized via a standard product computation.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be regular languages over Σ. Their intersection is defined as $\mathcal{L}_1 \cap \mathcal{L}_2 = \{x \mid x \in \mathcal{L}_1 \text{ and } x \in \mathcal{L}_2\}$. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the DFAs that accept $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. The *product* of $\mathcal{M}_1$ and $\mathcal{M}_2$ is $\mathcal{M}_1 \times \mathcal{M}_2$ where the language accepted by the DFA $\mathcal{M}_1 \times \mathcal{M}_2$ is equal to $\mathcal{L}_1 \cap \mathcal{L}_2$ (i.e., $\mathcal{L}(\mathcal{M}_1 \times \mathcal{M}_2) = \mathcal{L}_1 \cap \mathcal{L}_2$).

Definition B.1 (Classifier Modification) *Given a DFA encoding some classification criterion $\mathcal{M}_c$ and a DFA classifier $\mathcal{M}$, the modified classifier $\mathcal{M}'$ is the product of $\mathcal{M}$ and $\mathcal{M}_c$.*

Classifier modification ensures the enforcement of criterion $\mathcal{M}_c$ in $\mathcal{M}'$. However, such post-training modification could result in rejection of traces in the dataset that are labelled as positive examples of the class. Such modification can (and should) be verified against the dataset. Finally, modification criteria can be expressed directly in a DFA, or specified in a more natural form such as LTL.

B.5 Linear Temporal Logic

In Section 4 we proposed Linear Temporal Logic (LTL) as a candidate language for conveying explanations *to* humans or other agents, and for use *by* humans or other agents to express temporal properties that the agent might wish to add to the classifier or have verified. In what follows we review the basic syntax and semantics of LTL. Note that LTL formulae can be interpreted over either infinite or finite traces, with the finite interpretation requiring a small variation in the interpretation of formulae in the final state of the finite trace. Here we describe LTL interpreted over infinite traces noting differences as relevant.

LTL is a propositional logic language augmented with modal temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$), from which it is possible to define the well-known operators *always* ($\Box$), *eventually* ($\Diamond$), and *release* ($\mathcal{R}$). When interpreted over finite traces, a *weak next* ($\bullet$) operator is also utilized, and is equivalent to $\bigcirc$ when π is infinite. An LTL formula over a set of propositions $\mathcal{P}$ is defined inductively: a proposition in $\mathcal{P}$ is a formula, and if ψ and χ are formulae, then so are $\neg\psi$, $(\psi \wedge \chi)$, $(\psi \,\mathcal{U}\, \chi)$, $\bigcirc\psi$, and $\bullet\psi$.

The semantics of LTL are defined as follows. A trace π is a sequence of states, where each state is an element in $2^{\mathcal{P}}$. We denote the first state of π as s_1 and the i-th state of π as s_i; $|\pi|$ is the length of π (which is ∞ if π is infinite). We say that π satisfies φ ($\pi \models \varphi$, for short) iff $\pi, 1 \models \varphi$, where for every $i \geq 1$:

- $\pi, i \models p$, for a propositional variable $p \in \mathcal{P}$, iff $p \in s_i$,
- $\pi, i \models \neg\psi$ iff it is not the case that $\pi, i \models \psi$,
- $\pi, i \models (\psi \wedge \chi)$ iff $\pi, i \models \psi$ and $\pi, i \models \chi$,
- $\pi, i \models \bigcirc\varphi$ iff $i < |\pi|$ and $\pi, i+1 \models \varphi$,
- $\pi, i \models (\varphi_1 \,\mathcal{U}\, \varphi_2)$ iff for some j in $\{i, \ldots, |\pi|\}$, it holds that $\pi, j \models \varphi_2$ and for all $k \in \{i, \ldots, j-1\}$, $\pi, k \models \varphi_1$,
- $\pi, i \models \bullet\varphi$ iff $i = |\pi|$ or $\pi, i+1 \models \varphi$.

$\Diamond\varphi$ is defined as $(\mathsf{true}\,\mathcal{U}\,\varphi)$, $\Box\varphi$ as $\neg\Diamond\neg\varphi$, and $(\psi\,\mathcal{R}\,\chi)$ as $\neg(\neg\psi\,\mathcal{U}\,\neg\chi)$.

Given an LTL formula φ there exists an automaton $\mathcal{A}_\varphi$ that accepts a trace π iff $\pi \models \varphi$. It follows that, given a set of consistent LTL formulae, $\{\varphi_1, \ldots, \varphi_n\}$, there exists an automaton, $\mathcal{A}_\varphi$,where $\varphi = \bigwedge_i \varphi_i$, that accepts a trace π iff $\pi \models \varphi_i$, for all i. As noted in Section 2 an automaton defines a language—a set of words that are accepted by the automaton. We say that an automaton $\mathcal{A}$ satisfies an LTL formula, φ, $\mathcal{A} \models \varphi$ iff for every accepting trace, π_i of $\mathcal{A}$, $\pi_i \models \varphi$. Such satisfying LTL formulae provide another means of explaining the behaviour of a DFA classifier.

Depending on whether LTL formula, φ, is interpreted over finite or infinite traces, different types of automata are needed to capture φ. For the purposes of this paper, it is sufficient to know that DFAs are sufficiently expressive to capture any LTL formulae interpreted over finite traces, but only a subset (a large and useful subset) of LTL formulae interpreted over infinite traces.

C Experimental Evaluation

C.1 Experimental Setup

We first provide experimental details for each method used in our main set of experiments in Section 5. DISC, LSTM, and HMM used a validation set consisting of 20% of the training traces per class on all domains except MIT-AR. This was since MIT-AR consisted of very limited training data, and using a validation set worsened performance in all cases. We describe the specific modifications for each method below. Additionally, minor changes were made for our experiments on multi-label classification (described in C.5).

DISC (our approach) used Gurobi optimizer to solve the MILP formulation for learning DFAs. We set $q_{\max}$, the maximum possible number of states in a DFA, to 5 for Crystal Island and MIT-AR and 10 for all other domains along with a time limit of 15 minutes to learn each DFA. DISC also uses two regularization terms to prevent overfitting: a term penalizing the number of transitions between different states, with coefficient λ_t; and a term penalizing nodes not assigned to an absorbing state, with coefficient λ_a. We set $\lambda_a = 0.001$, and use a validation procedure to choose λ_t from 11 approximately evenly-spaced values (on a logarithmic scale) between 0.0001 and 10, inclusive. The model with maximum F_1-score on the validation set is selected. For MIT-AR, instead of using a validation set, we choose λ_t from a small set of evenly-spaced values ($\{3, 5.47, 10\}$) and select the model with highest training F_1-score.

The DFA-FT baseline utilized the full tree of observations (rather than the prefix tree used in DISC) and learned one DFA per label. A single positive and negative DFA state were designated, and any node in the tree whose suffixes were all positive or negative were assigned to the positive or negative state, respectively. Every other node of the tree was assigned to a unique DFA state and attached with the empirical (training) probability of a trace being positive, given that it transitions through that DFA state. To classify a trace in the presence of multiple classes, all $|\mathcal{C}|$ DFAs were run in parallel, and the class of the DFA with highest probability was returned.

Our LSTM model consisted of two LSTM layers, a linear layer mapping the final hidden state to labels, and a log-softmax layer. The LSTM optimized a negative log-likelihood objective using Adam optimizer [17], with equal weight assigned to each prefix of the trace (to encourage early prediction). We observed inferior performance overall when using one or four LSTM layers. The batch size was selected from $\{8, 32\}$, the size of the hidden state from $\{25, 50\}$, and the number of training epochs from $[1, 300]$ by choosing the model with the highest validation accuracy given full traces. For the MIT-AR dataset, the hyperparameters were hand-tuned to 8 for batch size, 25 for hidden dimension, and 75 epoches.

Our HMM model was based on an open-source Python implementation for unsupervised HMMs from Pomegranate[1]. We trained a separate HMM for each class, and classify a trace by choosing the HMM with highest probability. Each HMM was trained with the Baum-Welch algorithm using a stopping threshold of 0.001 and a maximum of 10^6 iterations. The validation set was used to select the number of discrete hidden states from $\{5, 10\}$ and a pseudocount (for smoothing) from $\{0, 0.1, 1\}$. For MIT-AR we hand-tuned these hyperparameters to 10 for the number of hidden states and 1 for the pseudocount.

[1] `https://pomegranate.readthedocs.io/en/latest/`

The n-gram models did not require validation. We used a smoothing constant $\alpha = 0.5$ to prevent estimating a probability of 0 for unseen sequences of observations.

C.2 Datasets

The StarCraft and Crystal Island datasets were obtained thanks to the authors [12, 15], while the malware datasets, ALFRED, and MIT-AR are publicly available[2][3][4].

Malware

The two malware datasets (BootCompleted, BatteryLow) were generated by Bernardi et al. (2019) by downloading and installing various malware applications with various intents (e.g., wiretapping, selling user information, advertisement, spam, stealing user credentials, ransom) on an Android phone. Each dataset reflects an Android operating system event (e.g., the phone's battery is at 50%) that is broadcasted system-wide (such that the broadcast also reaches every active application, including the running malware). Each family of malware is designed to react to a system event in a certain way, which can help distinguish it from the other families of malware (see Table 4 in [3] for the list of malware families used in the dataset).

A single trace in the dataset comprises a sequence of 'actions' performed by the malware application (e.g., the system call *clock_gettime*) in response to the Android system call in question, and labelled with the class label corresponding to the particular malware family.

StarCraft

The StarCraft dataset was constructed by Kantharaju et al. (2019) by using replay data of StarCraft games where various scripted agents were playing against one another. To this end, the real-time strategy testbed MicroRTS[5] was used. The scripted agents played in a 5-iterations round-robin tournament with the following agent types: *POLightRush, POHeavyRush, PORangedRush, POWorkerRush, EconomyMilitaryRush, EconomyRush, HeavyDefense, LightDefense, RangedDefense, WorkerDefense, WorkerRushPlusPlus*. Each agent competed against all other agents on various maps.

A replay for a particular game comprises a sequence of both players' actions, from which the authors extracted one labelled trace for each player. We label each trace with the agent type (e.g. *WorkerRushPlusPlus*) that generated the behaviour.

Crystal Island

Crystal Island is an educational adventure game designed for middle-school science students [12, 22], with the dataset comprising in-game action sequences logged from students playing the game. *"In Crystal Island, players are assigned a single high-level objective: solve a science mystery. Players interleave periods of exploration and deliberate problem solving in order to identify a spreading illness that is afflicting residents on the island. In this setting, goal recognition involves predicting the next narrative sub-goal that the player will complete as part of investigating the mystery"* [12]. Crystal Island is a particularly challenging dataset due to players interleaving exploration and problem solving which leads to noisy observation sequences.

A single trace in the dataset comprises a sequence of player actions, labelled with a single narrative sub-goal (e.g., *speaking with the camp's virus expert* and see Table 2 in [12]). Each observation in the trace includes one of 19 player action-types (e.g., testing an object using the laboratory's testing equipment) and one of 39 player locations. Each unique pair of action-type and location is treated as a distinct observation token.

ALFRED

ALFRED (Action Learning From Realistic Environments and Directives) is a benchmark for learning a mapping from natural language instructions and egocentric vision to sequences of actions for

[2] `https://github.com/mlbresearch/syscall-traces-dataset`

[3] `https://github.com/askforalfred/alfred/tree/master/data`

[4] `https://courses.media.mit.edu/2004fall/mas622j/04.projects/home/`

[5] `https://github.com/santiontanon/microrts`

household tasks. We generate our training set from the set of expert demonstrations in the ALFRED dataset which were produced by a classical planner given the high-level environment dynamics, encoded in PDDL [20]. Task-specific PDDL goal conditions (e.g., rinsing off a mug and placing it in the coffee maker) were then specified and given to the planner, which generated sequences of actions (plans) to achieve these goals. There are 7 different task types which we cast as the set of class labels $\mathcal{C}$ (see Figure 2 in [27]): *Pick & Place; Stack & Place; Pick Two & Place; Examine in Light; Heat & Place; Cool & Place; Clean & Place*.

A single trace in the dataset comprises a sequence of actions taken by the agent in the virtual home environment, labelled with one of the class labels described above (e.g., *Heat & Place*).

MIT Activity Recognition (MIT-AR)

MIT-AR was generated by Tapia et al. (2004) by collecting sensor data over a two week period from multiple sensors installed in a myriad of everyday objects such as drawers, refrigerators and containers. The sensors recorded opening and closing events related to these objects while the human subject carried out everyday activities. The resulting noisy sensor sequence data was manually labelled with various high-level daily activities in which the human subject was engaged. The activities in this dataset (which serve as the class labels in our experiments) include *preparing dinner*, *listening to music*, *taking medication* and *washing dishes*, and are listed in Table 5.3 in [31]. In total there are 14 activities.

A single trace in the dataset comprises a sequence of sensor recordings (e.g., *kitchen drawer interacted with* or *kitchen washing machine interacted with*), labelled with one of the class labels described above (e.g., *washing dishes*).

C.3 Additional Results

			Percent Accuracy given full observation traces					
Dataset	N	$\lvert\tau\rvert$	**DISC**	DFA-FT	LSTM	HMM	1-gram	2-gram
Crystal Island	893	52.9	78 ($\pm$1.2)	46 ($\pm$1.0)	**87** ($\pm$1.1)	57 ($\pm$1.2)	69 ($\pm$0.9)	57 ($\pm$1.1)
StarCraft	3872	14.8	43 ($\pm$0.4)	38 ($\pm$0.4)	**44** ($\pm$0.4)	38 ($\pm$0.6)	29 ($\pm$0.4)	37 ($\pm$0.4)
ALFRED	2520	7.5	**99** ($\pm$0.1)	94 ($\pm$0.3)	**99** ($\pm$0.1)	97 ($\pm$0.7)	83 ($\pm$0.3)	94 ($\pm$0.2)
MIT-AR	283	9.3	57 ($\pm$1.9)	36 ($\pm$1.9)	56 ($\pm$1.9)	45 ($\pm$2.1)	**66** ($\pm$2.0)	55 ($\pm$1.5)
BootCompleted	477	206.0	59 ($\pm$2.2)	**69** ($\pm$1.5)	65 ($\pm$1.3)	54 ($\pm$2.4)	46 ($\pm$2.8)	55 ($\pm$1.6)
BatteryLow	283	216.2	60 ($\pm$1.4)	**73** ($\pm$1.4)	70 ($\pm$1.4)	52 ($\pm$2.2)	35 ($\pm$1.3)	54 ($\pm$1.5)

Table 1: A summary of results from all domains (**DISC** is our approach). With respect to the full dataset, N is the total number of traces, and $|\tau|$ is the average length of a trace. Reported are the percentages of traces correctly classified given the full observation trace, with 90% confidence error in parentheses. Highest accuracy is bolded.

	(# DFA states, # state transitions)	
Dataset	**DISC**	DFA-FT
StarCraft	8.8, 37.2	170.4, 196.0
MIT-AR	3.0, 1.83	18.3, 103.4
Crystal Island	3.9, 26.2	166.6, 451.4
ALFRED	5.1, 9.0	26.8, 53.2
BootCompleted	9.7, 42.7	321.3, 391.5
BatteryLow	8.9, 27.2	325.1, 365.7

Table 2: The average number of DFA states (first), and the average number of state transitions (second) in learned models for **DISC** (ours) and DFA-FT over twenty runs, using the experimental procedure in Section C.1.

A summary of results over all datasets is reported in Table 1. We display the extensive results for each dataset in Figures 7, 8, 9. For each domain, we present a line plot displaying the Cumulative Convergence Accuracy (CCA) up to the maximum length of any trace and a bar plot displaying the PCA at 20%, 40%, 60%, 80%, and 100% of observations. Error bars report a 90% confidence interval over 30 runs.

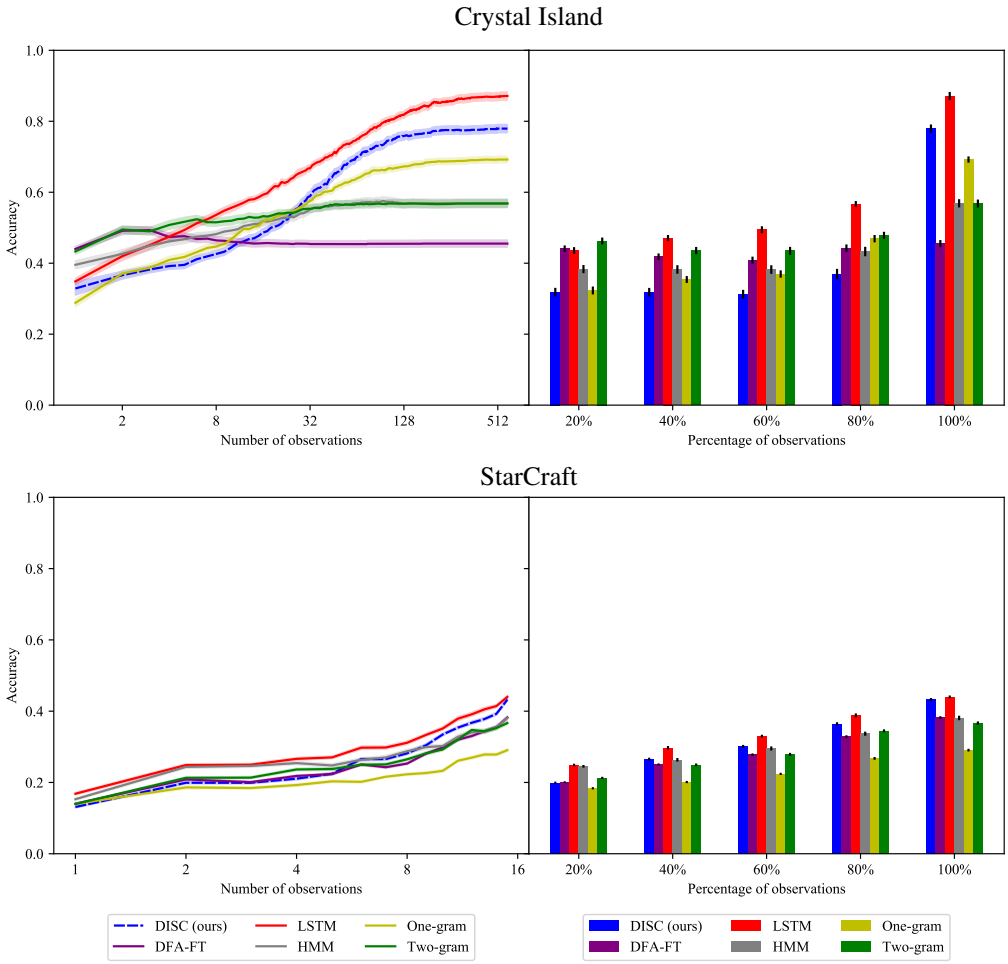

Figure 7: Results for the Crystal Island and StarCraft domains.

We further report in Table 2 the average number of states and transitions in learned DFAs for DISC and DFA-FT. DFAs learned using DISC were generally an order of magnitude smaller than DFAs learned using DFA-FT.

C.4 Early Classification

The two key problems in early prediction are: (1) to maximize accuracy given only a prefix of the sequence and (2) to determine a stopping rule for when to make a classification decision. (1) is not significantly different from vanilla sequence classification, thus, most work in early prediction focuses on (2). While many different stopping rules have been proposed in the literature, the correct choice should be task-dependent as it requires making a trade-off between accuracy and earliness. Furthermore, it is difficult to objectively compare early prediction models that may make decisions at different times. Our early classification experiment is designed to evaluate two essential criteria: the accuracy of early classification, and the accuracy of classifier confidence, while remaining independent of choice of stopping rule.

Thus, we expand upon the early classification setting briefly mentioned in Section 5 where an agent can make an irrevocable classification decision after any number of observations, but prefers to make a correct decision as early as possible. This is captured by a non-increasing utility function $U(t)$ for a correct classification. Note the agent can usually improve its chance of a correct prediction by waiting to see more observations. If the agent's predictive accuracy after t observations is $p(t)$, then to maximize expected utility, the agent should make a decision at time $t^* = \text{argmax}_t\{U(t)p(t)\}$. However, the agent only has access to its estimated confidence measures $\text{conf}(t) \approx p(t)$. Thus,

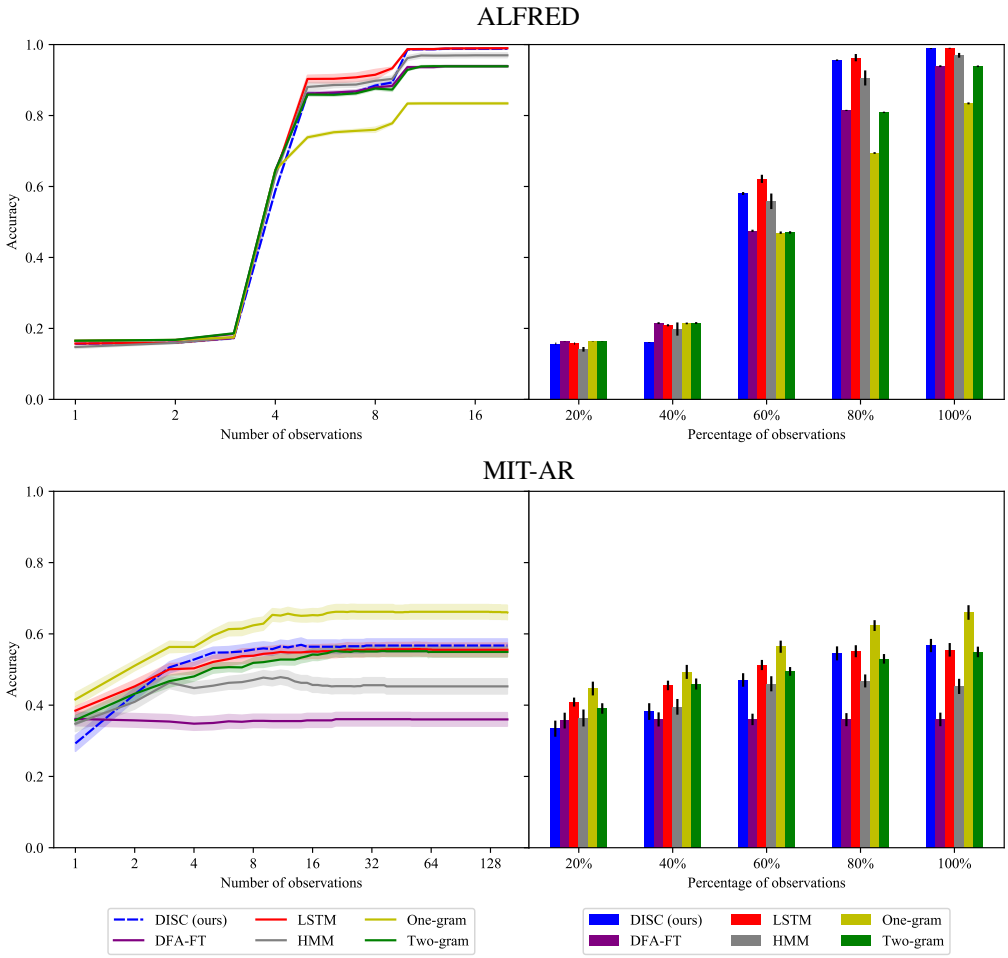

Figure 8: Results for the Alfred and MIT-AR domains.

	Average utility			
Dataset	**DISC**	LSTM	1-gram	2-gram
ALFRED	$0.840(\pm 0.014)$	$\mathbf{0.855}(\pm 0.003)$	$0.703(\pm 0.002)$	$0.792(\pm 0.003)$
StarCraft	$0.337(\pm 0.005)$	$\mathbf{0.341}(\pm 0.005)$	$0.194(\pm 0.004)$	$0.273(\pm 0.006)$
BootCompleted	$0.203(\pm 0.014)$	$\mathbf{0.218}(\pm 0.018)$	$0.113(\pm 0.003)$	$0.157(\pm 0.006)$

Table 3: Results for the early classification experiment. Average utility per trace over twenty runs is reported with 90% confidence error, with the best mean performance in each row in bold.

success in this task requires not only high classification accuracy, but also accurate confidence in one's own predictions.

We test this setting on a subset of domains, with utility function $U(t) = \max\{1 - \frac{t}{40}, 0\}$. We make the assumption that at time t, the classifier only has access to the first t observations, but has full access to the values of $\text{conf}(t')$ for all t' and can therefore choose the optimal decision time. We only consider baselines which produce a probability distribution over labels (DISC, LSTM, n-gram), defining the classifier's confidence to be the probability assigned to the predicted label (i.e. the most probable goal).

Results are shown in Table 3. DISC has a strong performance on each domain, only comparable by LSTM. This suggests the confidence produced by DISC accurately approximates its true predictive accuracy.

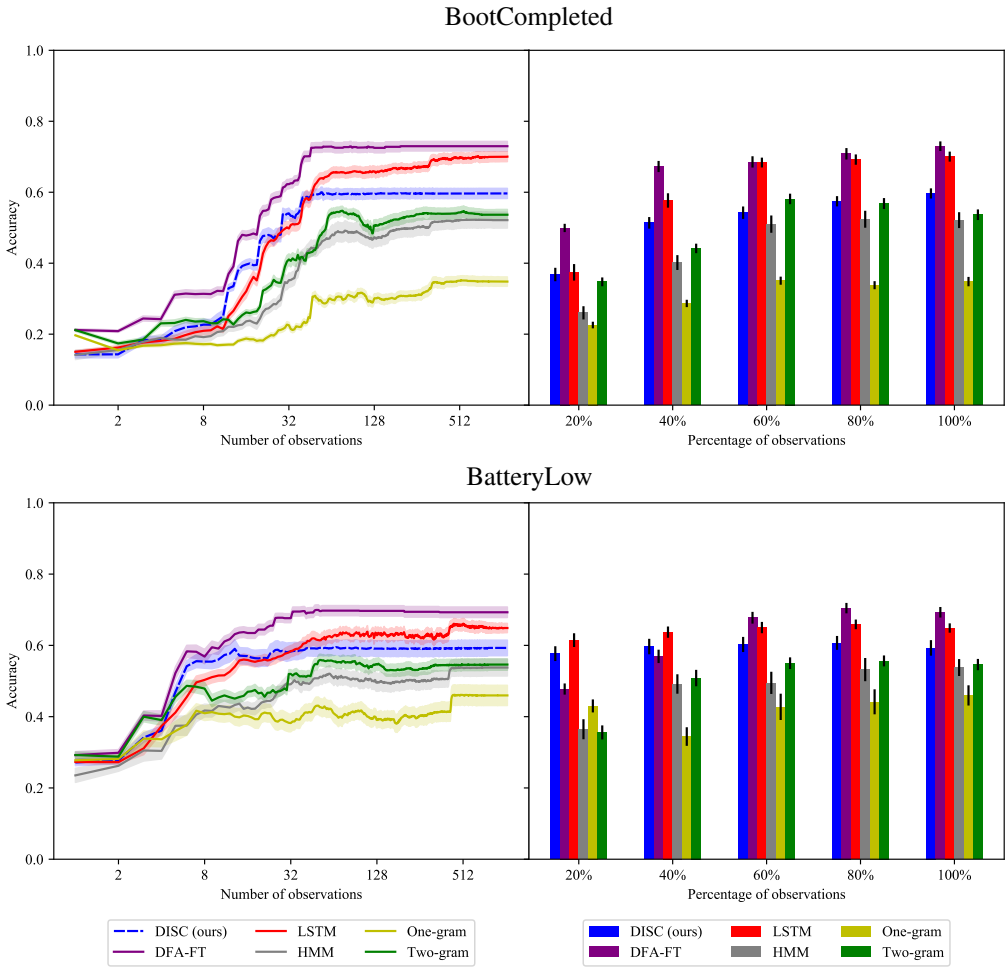

Figure 9: Results for the BootCompleted and BatteryLow domains.

C.5 Multi-label Classification

In the goal recognition datasets used in our work we assume agents pursue a single goal achieved by the sequence of actions encoded in the sequence of observations. However, often times an agent will pursue multiple goals concurrently, interleaving actions such that each action in a trace is aimed at achieving any one of multiple goals. For instance, if an agent is trying to make toast *and* coffee, the first action in their plan may be to fill the kettle with water, the second action may be to put the kettle on the stove, their third action might be to take bread out of the cabinet, and so on. We cast this generalization of the goal recognition task as a multi-label classification problem where each trace may have one or more class labels (e.g., *toast* and *coffee*).

We experiment with a synthetic kitchen dataset [13] where an agent is pursuing multiple goals and non-deterministically switching between plans to achieve them. A single trace in this dataset comprises actions performed by the agent in the kitchen environment in pursuit of multiple goals drawn from the set of possible goals. Each trace is labelled with multiple class labels corresponding to the goals achieved by the interleaved plans encoded in the trace. In total there are 7 goals the agent may be pursuing ($|\mathcal{C}| = 7$) and 25 unique observations ($|\Sigma| = 25$). The multi-goal kitchen dataset was obtained with thanks to the authors [13].

We modify DISC for this setting by directly using the independent outputs of the binary one-vs-rest classifiers—Bayesian inference is no longer necessary since we do not need to discriminate a single label. Precisely, for a given trace τ, we independently run all $|\mathcal{C}|$ DFA classifiers and return *all* classes for which the corresponding DFA accepts. We also disable the reweighting technique described in A.2 (i.e. by setting $\lambda^+ = \lambda^- = 1$) to focus on optimizing accuracy. We set DISC's hyperparameters

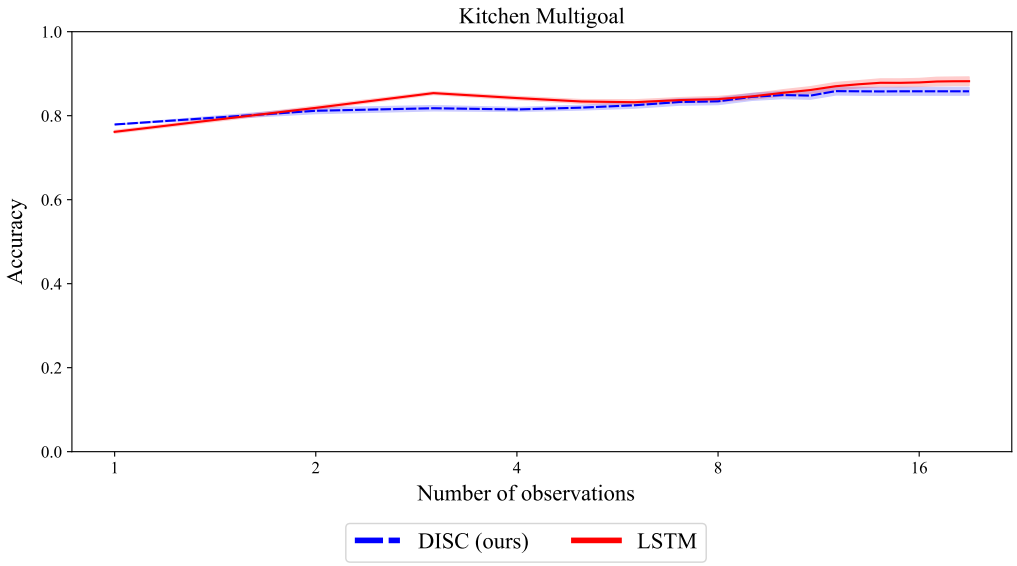

Figure 10: CCA for the Kitchen domain. Error bars represent a 90% confidence interval over 30 runs.

to $q_{\max} = 5, \lambda_a = 0.001, \lambda_t = 0.0003$. The LSTM baseline is modified to return a $|\mathcal{C}|$-dimensional output vector containing an independent probability for each class and is trained with a cross-entropy loss averaged over all dimensions. At test time, we predict all classes $c \in \mathcal{C}$ with probability greater than 0.5. We set the LSTM's hyperparameters to 8 for batch size, 25 for hidden dimension size, and 250 for number of epoches. Results are shown in Figure 10, where we report the mean accuracy, averaged over all goals, over 30 runs. DISC achieves similar performance to LSTM (c) on this task.