Interpretable Sequence Classification via Discrete Optimization (Technical Appendix)

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, 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()`.

Algorithm 1 Converting Training Data into Prefix Trees

```
1: function GET_PREFIX_TREE(𝒯, c⁺)
2:     r ← create_root_node()
3:     for (τ, c) ∈ 𝒯 do
4:         n ← r
5:         add_cost(n, c = c⁺, |τ|)
6:         for σ ∈ τ do
7:             if not has_child(n,σ) then
8:                 add_child(n,σ)
9:             end if
10:            n ← get_child(n,σ)
11:            add_cost(n, c = c⁺, |τ|)
12:        end for
13:    end for
14:    return r
15: end function
```

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.

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

$$s.t. \sum_{q \in Q} x_{n,q} = 1 \qquad \forall n \in N \qquad (1)$$

$$x_{r,0} = 1 \qquad (2)$$

$$\sum_{q' \in Q} \delta_{q,\sigma,q'} = 1 \qquad \forall q \in Q, \sigma \in \Sigma \qquad (3)$$

$$\delta_{q,\sigma,q} = 1 \qquad \forall q \in T, \sigma \in \Sigma \tag{4}$$

$$x_{p(n),q} + x_{n,q'} - 1 \leq \delta_{q,s(n),q'} \qquad \forall n \in N \setminus \{r\}, q \in Q, q' \in Q \tag{5}$$

$$c_n = \lambda^+ \sum_{q \in F} c^+(n) x_{n,q} + \lambda^- \sum_{q \in Q \setminus F} c^-(n) x_{n,q} \qquad \forall n \in N \tag{6}$$

$$e_{q,\sigma} = \sum_{q' \in Q \setminus \{q\}} \delta_{q,\sigma,q'} \qquad \forall q \in Q, \sigma \in \Sigma \tag{7}$$

$$t_n = \sum_{q \in Q \setminus T} x_{n,q} \qquad \forall n \in N \tag{8}$$

$$x_{n,q} \in \{0,1\} \qquad \forall n \in N, q \in Q \tag{9}$$

$$\delta_{q,\sigma,q'} \in \{0,1\} \qquad \forall q \in Q, \sigma \in \Sigma, q' \in Q \tag{10}$$

$$c_n \in \mathbb{R} \qquad \forall n \in N \tag{11}$$

$$e_{q,\sigma} \in \mathbb{R} \qquad \forall q \in Q, \sigma \in \Sigma \tag{12}$$

$$t_n \in \mathbb{R} \qquad \forall n \in N \tag{13}$$

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

Recall 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

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

B Interpretability

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 - see Definition 4.1 for details) 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 τ = (**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.

Malware

We present two DFAs learned via DISC from the real-world *malware* datasets. Figure 4 depicts a DFA classifier for Battery-Low 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 τ = (*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 5 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) τ = (*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 $q_{\max} = 5$.

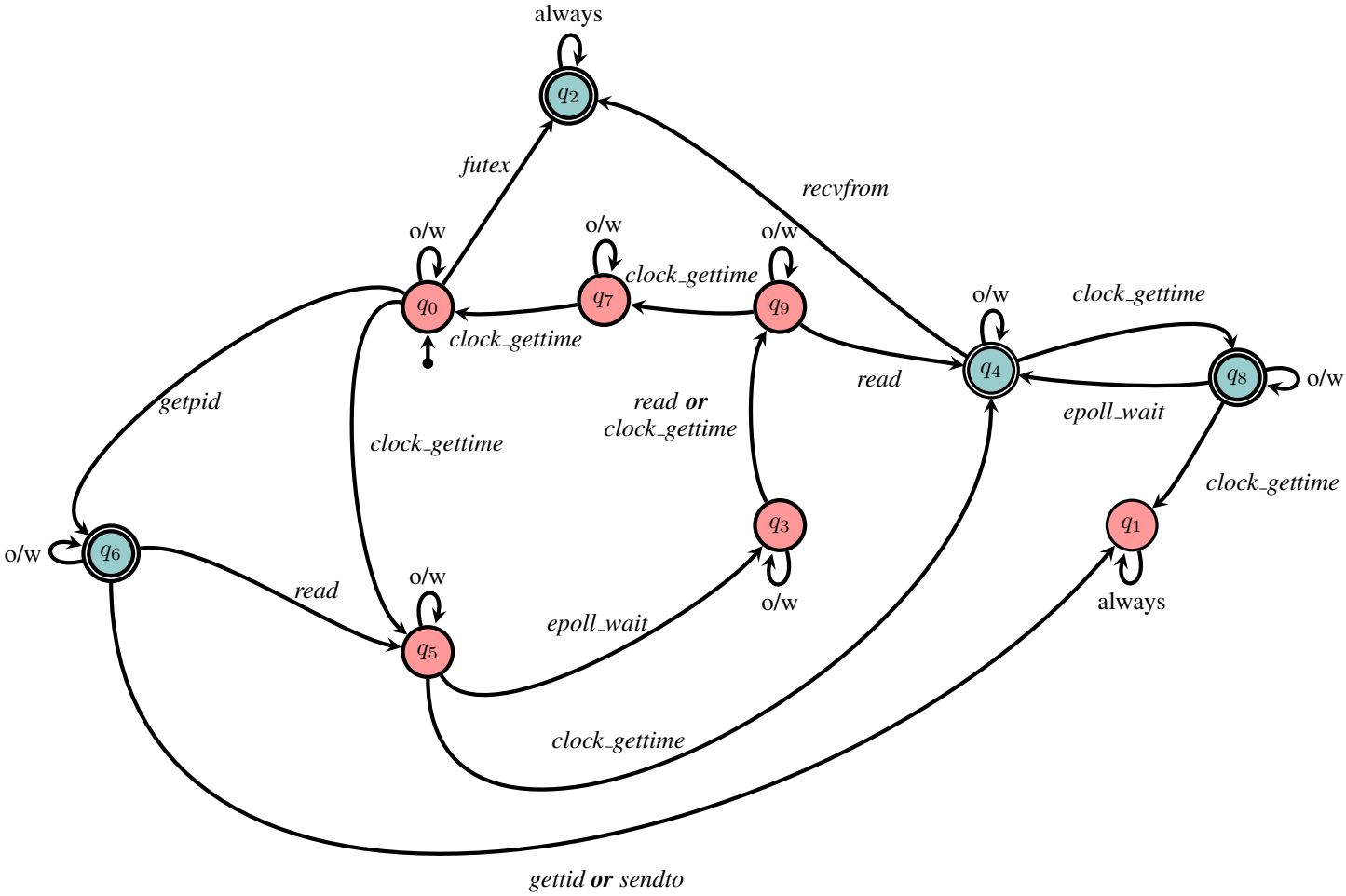

Figure 4: A DFA learned in our experiments from the BatteryLow dataset by limiting the maximum number of states to 10. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

Crystal Island

Figure 6 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 τ = (*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.

StarCraft

Figure 7 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 τ = (*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

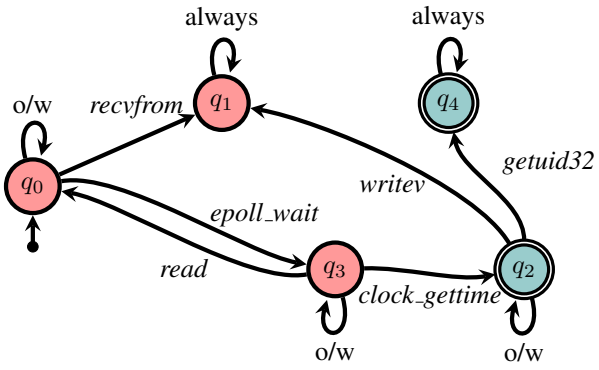

Figure 5: A DFA learned from the BootCompleted dataset by limiting the maximum number of states to 5. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that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

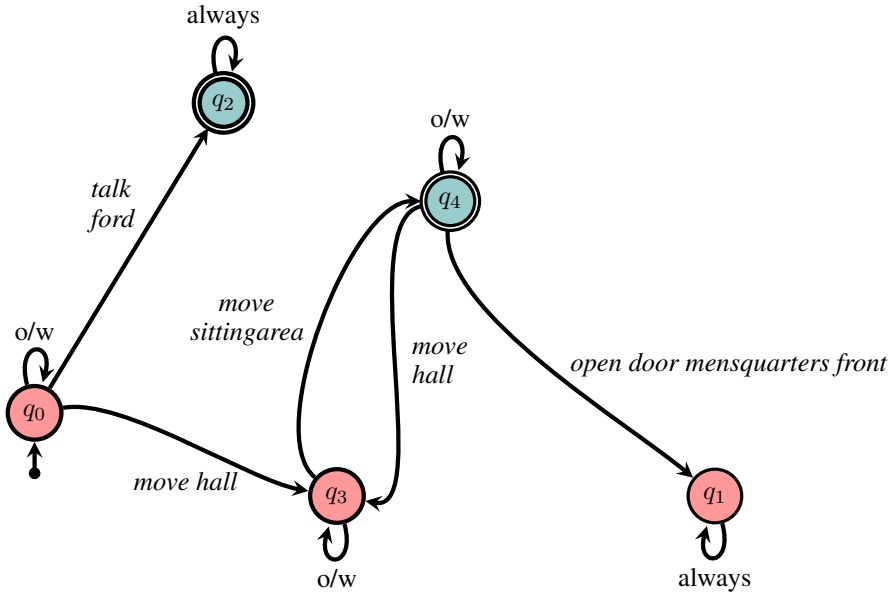

Figure 6: A DFA learned from the Crystal Island dataset, limiting the maximum number of states to 5. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

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.

B.3 Linear Temporal Logic

In Section 4 of our paper, 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 (Pnueli 1977). 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 $\mathcal{P}$

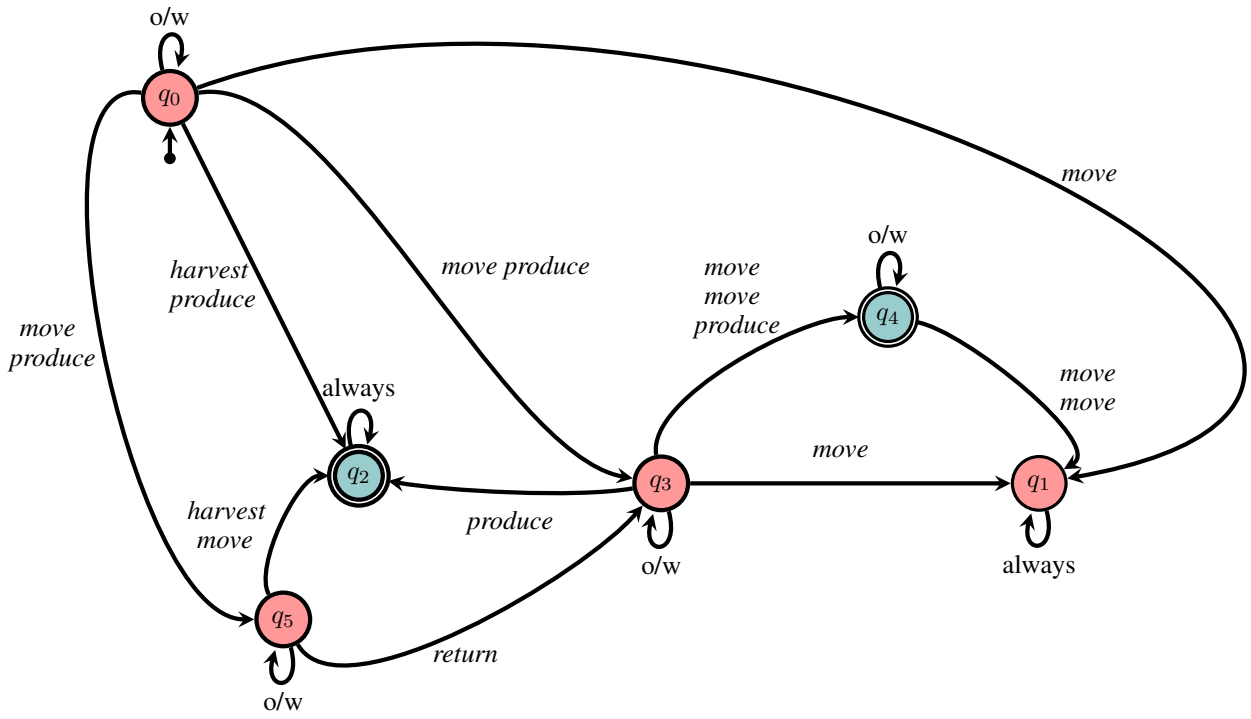

Figure 7: A DFA learned from the StarCraft dataset by limiting the maximum number of states to 10. A decision is provided after each new observation based on the current state: yes for the blue accepting state, and no for the red, non-accepting states. "o/w" (otherwise) stands for all symbols that do not appear on outgoing edges from a state. "always" stands for all symbols.

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 is 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 $\pi \models \varphi$. 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 (Kingma and Ba 2014), 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[2]. 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.

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, while the malware datasets, ALFRED, and MIT-AR are publicly available[3][4][5].

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 (Bernardi et al. 2019) 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.

[2]https://pomegranate.readthedocs.io/en/latest/

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

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

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

StarCraft

The StarCraft dataset was constructed by Kantharaju, Ontañón, and Geib (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[6] 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 (Ha et al. 2011; Min et al. 2016), 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"* (Ha et al. 2011). 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 (Ha et al. 2011)). 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 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 (McDermott et al. 1998). 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 (Shridhar et al. 2020)): *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, Intille, and Larson (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 (Tapia, Intille, and Larson 2004). 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

We display the extensive results from the main paper in Figures 8, 9, 10. 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.

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.

[6]https://github.com/santiontanon/microrts

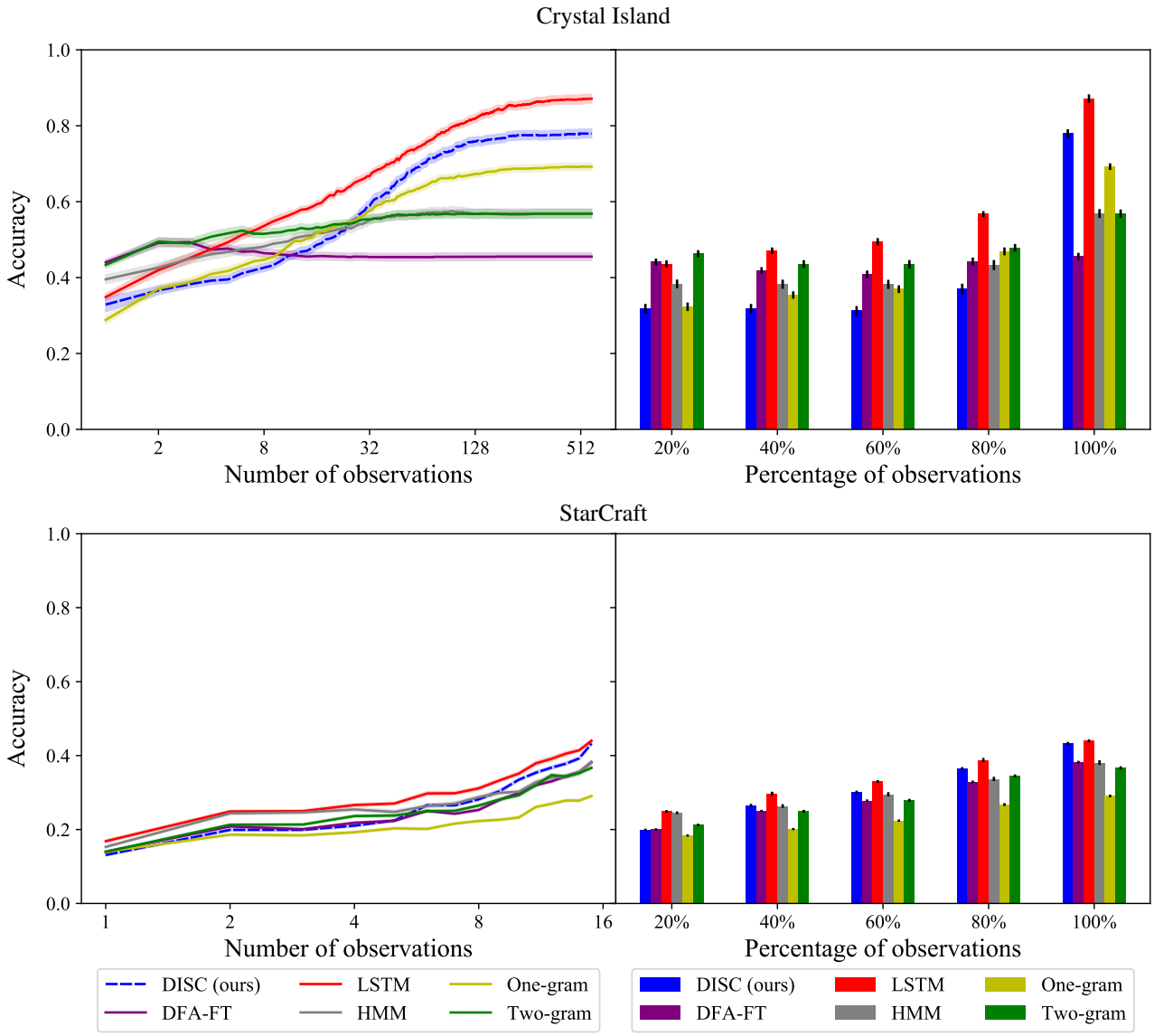

Figure 8: Results for the Crystal Island and StarCraft domains.

	(# 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.

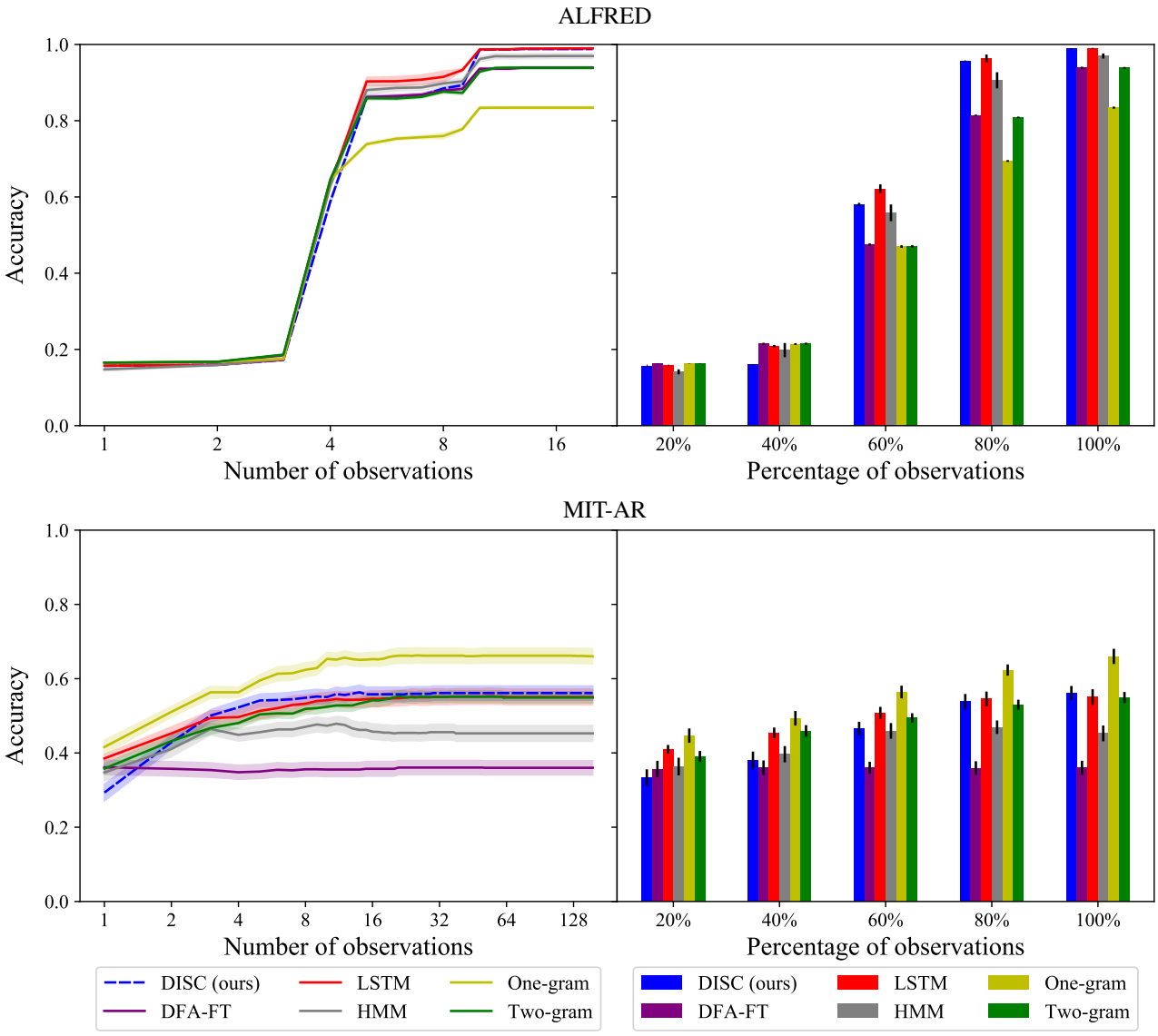

Figure 9: Results for the Alfred and MIT-AR domains.

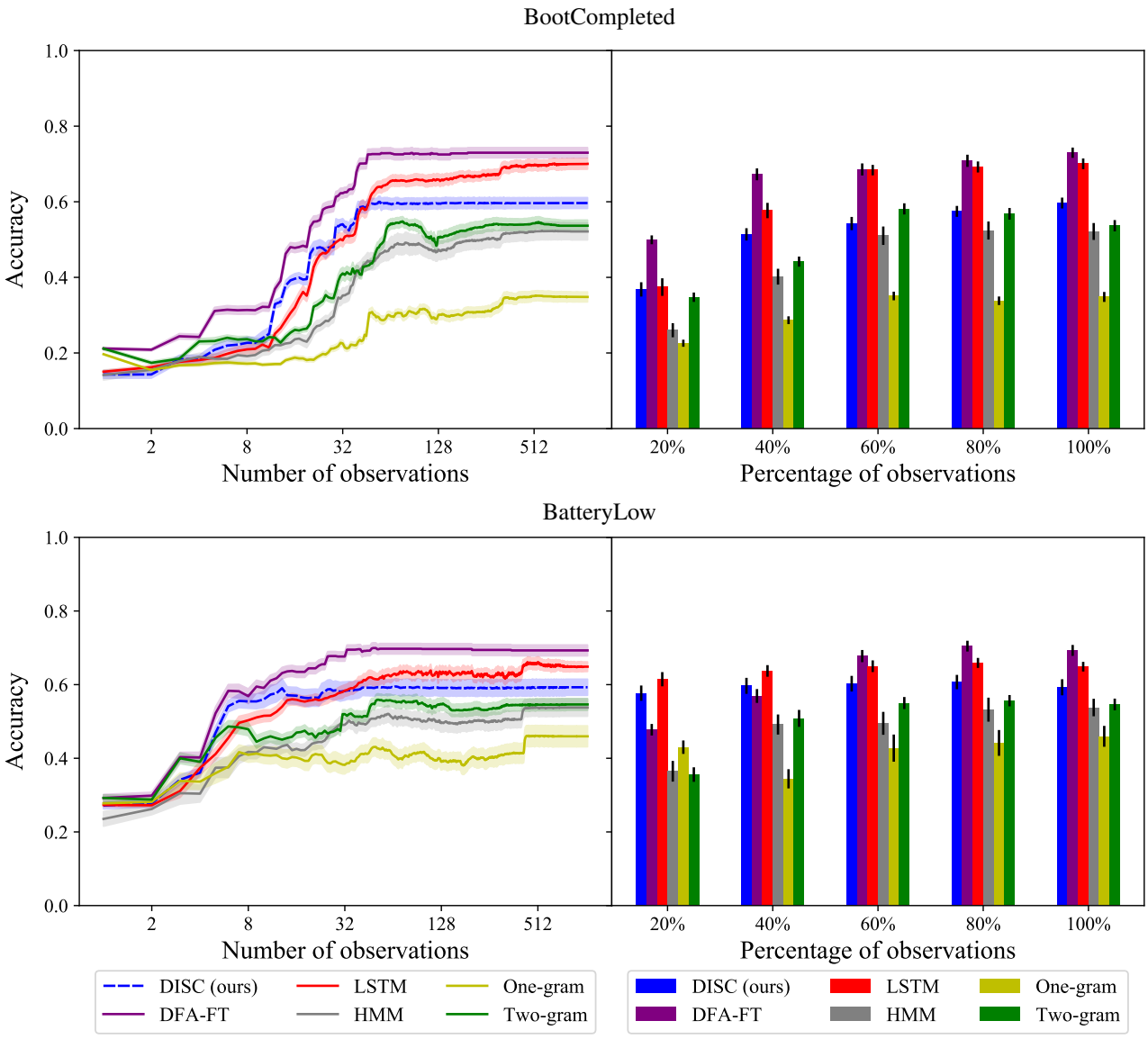

Figure 10: Results for the BootCompleted and BatteryLow domains.

	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.

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 5.2 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, 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.

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 (Harman and Simoens 2020) 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 (Harman and Simoens 2020).

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 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 11, where we report the mean accuracy, averaged over all goals, over 30 runs. DISC achieves similar performance to LSTM (c) on this task.

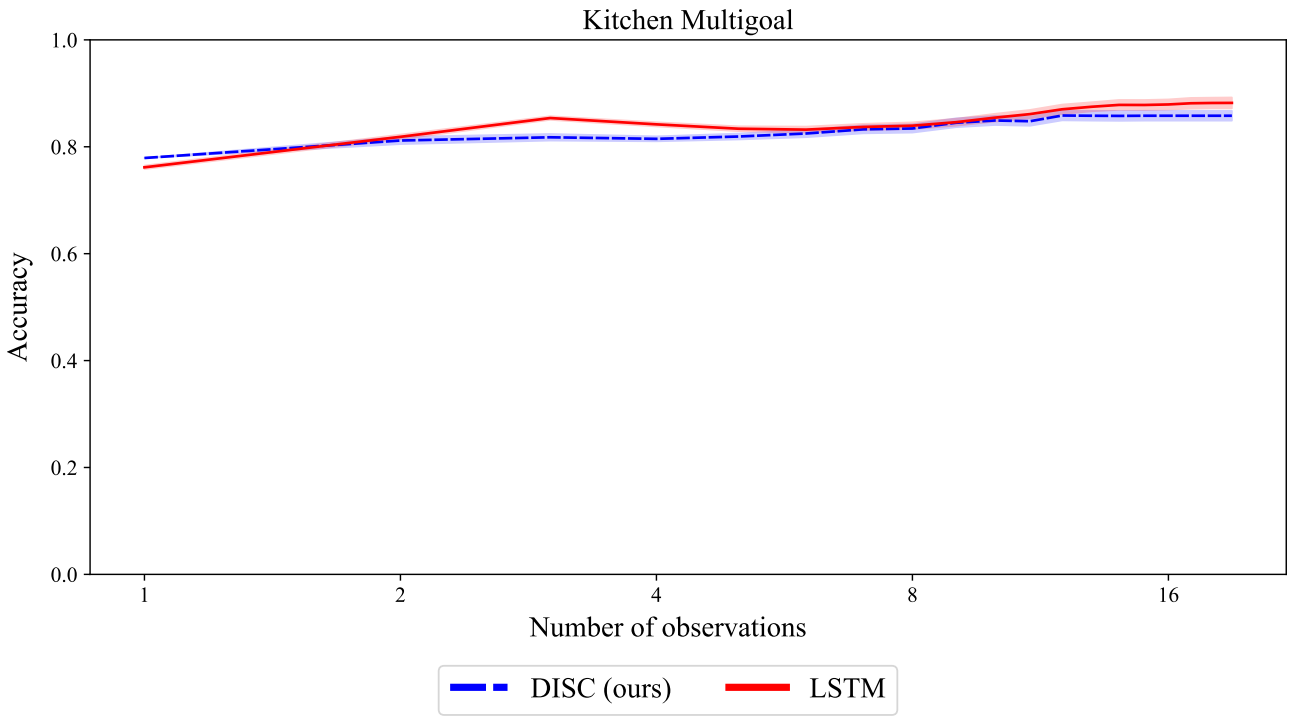

Figure 11: CCA for the Kitchen domain. Error bars represent a 90% confidence interval over 30 runs.