Learning MDL Logic Programs from Noisy Data

Céline Hocquette[1], **Andreas Niskanen**[2], **Matti Järvisalo**[2], **Andrew Cropper**[1]
[1]University of Oxford
[2]University of Helsinki
celine.hocquette@cs.ox.ac.uk, andreas.niskanen@helsinki.fi, matti.jarvisalo@helsinki.fi, andrew.cropper@cs.ox.ac.uk

Abstract

Many inductive logic programming approaches struggle to learn programs from noisy data. To overcome this limitation, we introduce an approach that learns minimal description length programs from noisy data, including recursive programs. Our experiments on several domains, including drug design, game playing, and program synthesis, show that our approach can outperform existing approaches in terms of predictive accuracies and scale to moderate amounts of noise.

1 Introduction

The goal of inductive logic programming (ILP) (Muggleton 1991) is to induce a logic program (a set of logical rules) that generalises training examples and background knowledge. A common criticism of ILP is that it cannot handle noisy data (Evans and Grefenstette 2018; Cucala, Grau, and Motik 2022). This criticism is unfounded: most ILP approaches can learn from noisy data (Cropper and Dumancic 2022). For instance, set-covering approaches (Muggleton 1995; Srinivasan 2001; Ahlgren and Yuen 2013; Zeng, Patel, and Page 2014; De Raedt et al. 2015) search for rules that generalise a subset of the examples.

Although most ILP approaches can learn from noisy data, they struggle to learn recursive programs and perform predicate invention, two important features when learning complex algorithms (Lin et al. 2014; Cropper and Muggleton 2019). Moreover, they are not guaranteed to learn optimal programs, such as textually minimal programs, and tend to overfit.

Recent approaches overcome these limitations and can learn recursive and textually minimal programs (Corapi, Russo, and Lupu 2011; Kaminski, Eiter, and Inoue 2019) and perform predicate invention (Muggleton, Lin, and Tamaddoni-Nezhad 2015; Purgal, Cerna, and Kaliszyk 2022). However, these approaches struggle to learn from noisy data because they search for a program that strictly generalises all the positive and none of the negative examples.

In this paper, our goal is to learn recursive programs and support predicate invention in a noisy setting. Following Cropper and Hocquette (2023), we first search for small programs that generalise a subset of the examples. We then search for a combination of these smaller programs to form a larger program. Cropper and Hocquette (2023) search for a combination that strictly generalises all the positive and none of the negative examples, i.e. they cannot learn from noisy data. By contrast, we relax this condition to learn from noisy data. To avoid overfitting, we search for a combination that trades off model complexity (program size) and data fit (training accuracy). To do so, we use the minimal description length (MDL) principle (Rissanen 1978). In other words, we introduce an approach that learns MDL programs from noisy data.

To explore our idea, we build on *learning from failures* (LFF) (Cropper and Morel 2021). LFF frames the ILP problem as a constraint satisfaction problem (CSP), where each solution to the CSP represents a program (a hypothesis). The goal of a LFF learner is to accumulate constraints to restrict the hypothesis space (the set of all hypotheses) and thus constrain the search. We use LFF to explore our idea because it can learn recursive programs and perform predicate invention. We build on LFF by learning MDL programs from noisy examples. We introduce constraints which are optimally sound in that they do not prune MDL programs. To find an MDL combination, we use a maximum satisfiability (MaxSAT) solver (Bacchus, Järvisalo, and Martins 2021).

Novelty and contributions The main novelty of this paper is the idea of learning small programs from noisy examples and using a MaxSAT solver to find an MDL combination. The benefits, which we show on diverse domains, are (i) the ability to learn complex programs from noisy examples, and (ii) improved performance compared to existing approaches.

Overall, our contributions are:

1. We introduce MAXSYNTH, which learns MDL programs from noisy examples, including recursive programs.
2. We introduce constraints for this noisy setting and prove that they are optimally sound (Propositions 1 and 2).
3. We prove the correctness of MAXSYNTH, i.e. that it always learns an MDL program (Theorem 1).
4. We experimentally show on multiple domains, including drug design, game playing, and program synthesis, that MAXSYNTH can (i) substantially improve predictive accuracies compared to other systems, and (ii) scale to moderate amounts of noise (30%). We also show that our noisy constraints can reduce learning times by 99%.

2 Related Work

ILP. Most ILP approaches support noise (Quinlan 1990; Muggleton 1995; McCreath and Sharma 1997; Blockeel and De Raedt 1998; Srinivasan 2001; Oblak and Bratko 2010; Ahlgren and Yuen 2013; Zeng, Patel, and Page 2014; De Raedt et al. 2015). However, these approaches do not support predicate invention, struggle to learn recursive programs, and are not guaranteed to learn an MDL program. Recent approaches can learn textually minimal and recursive programs but are not robust to noisy examples (Corapi, Russo, and Lupu 2011; Muggleton, Lin, and Tamaddoni-Nezhad 2015; Kaminski, Eiter, and Inoue 2019; Cropper and Morel 2021; Dai and Muggleton 2021; Purgal, Cerna, and Kaliszyk 2022). There are two notable exceptions. δILP (Evans and Grefenstette 2018) frames the ILP problem as a differentiable neural architecture and is robust to noisy data. NOISYPOPPER (Wahlig 2022) can learn MDL and recursive programs from noisy examples. However, these approaches can only learn programs with a small number of small rules. For instance, δILP cannot learn programs with more than a few rules and can only use binary relations. By contrast, MAXSYNTH can learn MDL programs with many rules and any arity relation.

Rule selection. Many systems formulate the ILP problem as a rule selection problem (Corapi, Russo, and Lupu 2011; Kaminski, Eiter, and Inoue 2019; Si et al. 2019; Raghothaman et al. 2020; Evans et al. 2021; Bembenek, Greenberg, and Chong 2023). These approaches precompute every possible rule in the hypothesis space and then search for a subset that entails all the positive and no negative examples. Some approaches relax this requirement to find a subset with the best coverage using solver optimisation (Law 2022; Evans et al. 2021) or numerical methods (Si et al. 2019). Precomputing all possible rules prevent these approaches from learning rules with a large number of literals. By contrast, our approach does not rely on exhaustive precomputation.

Sampling. Sampling can mitigate noise. Raychev et al. (2016) pair a data sampler which selects representative subsets of the data with a regularised program generator to avoid overfitting. METAGOL$_{nt}$ (Muggleton et al. 2018) finds hypotheses consistent with randomly sampled subsets of the training examples and evaluates each resulting program on the remaining training examples. METAGOL$_{nt}$ needs as input a parameter about the noise level. By contrast, MAXSYNTH does not need a user-provided noise level parameter and is guaranteed to learn an MDL program.

Rule mining. AMIE+ (Galárraga et al. 2015) learns rules from noisy knowledge bases. However, AMIE+ can only use unary and binary relations, so it cannot be used on most of the datasets in our experiments, which require relations of arity greater than two. By contrast, MAXSYNTH can learn programs with relations of any arity.

MDL. Several approaches use cost functions based on MDL (Quinlan 1990; Muggleton 1995; Srinivasan 2001; Huang and Pearce 2007). However, they are not guaranteed to find a program that minimises this cost function because they greedily learn a single rule at a time. By contrast, MAXSYNTH learns a global MDL program. Jain et al. (2021) learn propositional CNF using MDL. By contrast, we learn first-order theories.

3 Problem Setting

We describe our problem setting. We assume familiarity with logic programming (Lloyd 2012) but have included a summary in the appendix.

3.1 Learning From Failures

We use the LFF setting. A *hypothesis* is a definite program with the least Herbrand model semantics. A *hypothesis space* $\mathcal{H}$ is a set of hypotheses. LFF uses *hypothesis constraints* to restrict the hypothesis space. Let $\mathcal{L}$ be a meta-language that defines hypotheses. For instance, consider a language with two literals *h_lit/3* and *b_lit/3* which represent *head* and *body* literals respectively. With this language, we denote the rule *last(A,B)* $\leftarrow$ *tail(A,C), head(C,B)* as the set of literals $\{$*h_lit(0,last,(0,1)), b_lit(0,tail,(0,2)), b_lit(0,head,(2,1))*$\}$. The first argument of each literal is the rule index, the second is the predicate symbol, and the third is the literal variables, where *0* represents *A*, *1* represents *B*, etc. A *hypothesis constraint* is a constraint (a headless rule) expressed in $\mathcal{L}$. Let C be a set of hypothesis constraints written in a language $\mathcal{L}$. A hypothesis is *consistent* with C if when written in $\mathcal{L}$ it does not violate any constraint in C. We denote as $\mathcal{H}_C$ the subset of the hypothesis space $\mathcal{H}$ which does not violate any constraint in C.

We define a LFF input:

Definition 1 (LFF input). A *LFF input* is a tuple $(E, B, \mathcal{H}, C, cost)$ where $E = (E^+, E^-)$ is a pair of sets of ground atoms denoting positive (E^+) and negative (E^-) examples, B is a definite program denoting background knowledge, $\mathcal{H}$ is a hypothesis space, C is a set of hypothesis constraints, and *cost* is a function that measures the cost of a hypothesis.

We define a solution to a LFF input in the non-noisy setting:

Definition 2 (Non-noisy solution). Given a LFF input $(E, B, \mathcal{H}, C, cost)$, where $E = (E^+, E^-)$, a hypothesis $h \in \mathcal{H}_C$ is a *non-noisy solution* when h is *complete* ($\forall e \in E^+,\ B \cup h \models e$) and *consistent* ($\forall e \in E^-,\ B \cup h \not\models e$).

A hypothesis that is not a non-noisy solution is a *failure*. A LFF learner builds constraints from failures to restrict the hypothesis space. For instance, if a hypothesis h is inconsistent (entails a negative example), a generalisation constraint prunes generalisations of h as they are also inconsistent.

In the non-noisy setting, a cost function only takes as input a hypothesis, i.e. they are of the type $cost : \mathcal{H} \mapsto \mathbb{N}$. For instance, the cost of a hypothesis is typically measured as its size (the number of literals in the hypothesis). An *optimal* non-noisy solution minimises the cost function:

Definition 3 (Optimal non-noisy solution). Given a LFF input $(E, B, \mathcal{H}, C, cost)$, a hypothesis $h \in \mathcal{H}_C$ is an *optimal* non-noisy solution when (i) h is a non-noisy solution, and (ii) $\forall h' \in \mathcal{H}_C$, where h' is a non-noisy solution, $cost(h) \leq cost(h')$.

3.2 Noisy Learning From Failures

A non-noisy solution must entail all the positive and none of the negative examples. To tolerate noise, we relax this requirement. We generalise a LFF input to allow a cost function to also take as input background knowledge B and examples E, i.e. cost functions of the type $cost_{B,E} : \mathcal{H} \mapsto \mathbb{N}$. In our noisy setting, any hypothesis $h \in \mathcal{H}$ is a noisy solution. An *optimal* noisy solution minimises the cost function:

Definition 4 (**Optimal noisy solution**). Given a noisy input $(E, B, \mathcal{H}, C, cost_{B,E})$, a hypothesis $h \in \mathcal{H}_C$ is an *optimal* noisy solution when $\forall h' \in \mathcal{H}_C$, $cost_{B,E}(h) \leq cost_{B,E}(h')$.

3.3 Minimal Description Length

Our noisy LFF setting generalises the LFF setting to allow for different cost functions. A challenge in machine learning is choosing a suitable cost function. According to complexity-based induction, the best hypothesis is the one that minimises the number of bits required to communicate the examples (Conklin and Witten 1994). This concept corresponds to the hypothesis with minimal description complexity (Rissanen 1978)[1], where the idea is to trade off the complexity of a hypothesis (its size) with the fit to the data (training accuracy).

We use MDL as our cost function. To define it, we use the terminology of Conklin and Witten (1994). The MDL principle states that the most probable hypothesis h for the data E is the one that minimises the complexity $L(h|E)$ of the hypothesis given the data. The MDL principle can be expressed as finding a hypothesis that minimises $L(h)+L(E|h)$, where $L(h)$ is the syntactic complexity of a hypothesis h and $L(E|h)$ is the complexity of the examples when coded using h. We evaluate $L(h)$ with the function $size : \mathcal{H} \mapsto \mathbb{N}$, which measures the size of a hypothesis h as the number of literals in it. In a probabilistic setting, $L(E|h)$ is the log-likelihood of the data with respect to the hypothesis h. However, there is debate about how to interpret $L(E|h)$ in a logical setting. For instance, Muggleton, Srinivasan, and Bain (1992) use an encoding based on Turing machines (a proof complexity measure). We evaluate $L(E|h)$ as the cost of sending the exceptions to the hypothesis, i.e. the number of false positives $fp_{E,B}(h)$ (simply $fp(h)$) and false negatives $fn_{E,B}(h)$ (simply $fn(h)$). We define our MDL cost function:

Definition 5 (**MDL cost function**). Given examples E and background knowledge B, the MDL cost of a hypothesis $h \in \mathcal{H}$ is $cost_{B,E}(h) = size(h) + fn_{E,B}(h) + fp_{E,B}(h)$.

In other words, the MDL cost of a hypothesis h is the number of literals in h plus the number of false positives and false negatives of h on the training data[2].

In this paper, an *optimal noisy solution* refers to an optimal noisy solution with our MDL cost function.

3.4 Noisy Constraints

A LFF learner builds constraints from failures to restrict the hypothesis space. The existing constraints for LFF are intolerant to noise. For instance, if a hypothesis h is inconsistent, a non-noisy generalisation constraint prunes generalisations of h as they are also inconsistent. However, in a noisy setting, a generalisation of h might have a lower MDL cost. Therefore, the existing constraints can prune optimal noisy solutions from the hypothesis space.

To overcome this limitation, we introduce constraints that tolerate noise. These constraints are optimally sound for the noisy setting because they do not prune optimal noisy solutions from the hypothesis space. Due to space limitations, we only describe one specialisation and one generalisation constraint. The appendix contains a description of three other constraints. All the proofs are in the appendix.

Let h_1 be a hypothesis with $tp(h_1)$ true positives and h_2 be a specialisation of h_1. Then h_2 has at most $tp(h_1)$ true positives. Therefore, if $size(h_2) > tp(h_1)$ then the size of h_2 is greater than the number of positive examples it covers so h_2 cannot be in an optimal noisy solution:

Proposition 1 (**Noisy specialisation constraint**). Let h_1 be a hypothesis, h_2 be a specialisation of h_1, and $size(h_2) > tp(h_1)$. Then h_2 cannot be in an optimal noisy solution.

Similarly, let h_1 be a hypothesis with $fp(h_1)$ false positives and h_2 be a generalisation of h_1. Then h_2 has at least $fp(h_1)$ false positives and a cost of at least $fp(h_1) + size(h_2)$. We show that the cost of h_2 is greater than the cost of the empty hypothesis when $size(h_2) \geq |E^+| - fp(h_1)$:

Proposition 2 (**Noisy generalisation constraint**). Let h_1 be a hypothesis, h_2 be a generalisation of h_1, and $size(h_2) \geq |E^+| - fp(h_1)$. Then h_2 cannot be in an optimal noisy solution.

In the next section, we introduce MAXSYNTH which uses these optimally sound noisy constraints to learn programs.

4 Algorithm

We now describe our MAXSYNTH algorithm. We first describe POPPER (Cropper and Morel 2021; Cropper and Hocquette 2023), which MAXSYNTH builds on.

POPPER. POPPER takes as input background knowledge, positive and negative training examples, and a maximum hypothesis size. POPPER starts with an answer set programming (ASP) program $\mathcal{P}$. Each model (answer set) of $\mathcal{P}$ corresponds to a hypothesis (a definite program). POPPER uses a generate, test, combine, and constrain loop to find a textually minimal non-noisy solution. In the generate stage, POPPER uses Clingo (Gebser et al. 2014), an ASP system, to search for a model of $\mathcal{P}$ for increasing hypothesis sizes. If there is no model, POPPER increments the hypothesis size and loops again. If there is a model, POPPER converts it to a hypothesis h. In the test stage, POPPER uses Prolog to test h on the examples. If h is a non-noisy solution, POPPER returns it. If h covers at least one positive example and no negative examples, POPPER adds h to a set of promising programs. In the combine stage, POPPER searches for a combination (a union) of promising programs that covers as many positive examples as possible and is minimal in size. If POPPER finds a combination, it sets the combination as the best solution so far and updates the maximum hypothesis size. A combination may not cover all the positive examples (POPPER allows

[1]Selecting an MDL hypothesis is equivalent to selecting a hypothesis with the maximum Bayes' posterior probability (Muggleton and De Raedt 1994).

[2]It is straightforward to extend our MDL cost function to $\alpha size(h) + \beta fn_{E,B}(h) + \gamma fp_{E,B}(h)$ where α, β, and γ are positive integer weights. Our constraints easily generalise to this setting, as does our MaxSAT encoding.

Algorithm 1: MaxSynth

```
1  def maxsynth(bk, pos, neg):
2    cons, promising, best_solution = {}, {}, {}
3    size, max_mdl = 1, len(pos)
4    while size $\leq$ max_mdl:
5      h = generate(cons, size)
6      if h == UNSAT:
7        size += 1
8        continue
9      tp, fn, fp = test(pos, neg, bk, h)
10     h_mdl = fn+fp+size(h)
11     if h_mdl < max_mdl:
12       best_solution = h
13       max_mdl = h_mdl-1
14     if tp>0 and not_rec(h) and not_pi(h):
15       promising += h
16       combi = combine(promising, max_mdl)
17       if combi != UNSAT:
18         best_solution = combi
19         tp, fn, fp = test(pos, neg, bk, combi)
20         max_mdl = fn+fp+size(combi)-1
21     cons += constrain(h, fn, fp)
22   return best_solution
```

false positives) but it cannot cover any negative examples (Popper is intolerant to false negatives). In the constrain stage, Popper uses h to build hypothesis constraints (represented as ASP constraints). Popper adds these constraints to $\mathcal{P}$ to prune models and thus prune the hypothesis space. For instance, if h is inconsistent, Popper builds a generalisation constraint to prune the generalisations of h. Popper repeats this loop until it finds a textually minimal non-noisy solution or there are no more hypotheses to test.

4.1 MaxSynth

MaxSynth (Algorithm 1) is similar to Popper except for a few key differences. Popper searches for the smallest hypothesis that entails all the positive and none of the negative examples, i.e. it is intolerant to noisy data. By contrast, MaxSynth returns an MDL hypothesis, i.e. it is tolerant to noisy data. To find an MDL hypothesis, MaxSynth differs by (i) also saving inconsistent programs as promising programs, (ii) finding an MDL combination in the combine stage, and (iii) using noise-tolerant constraints to prune non-MDL programs. We describe these differences in turn.

Promising Programs Popper only saves consistent programs as promising programs. Popper is, therefore, intolerant to false negative training examples. To handle noise, MaxSynth relaxes this requirement and saves programs which cover at least one positive example as promising programs (line 15), even if they are inconsistent. MaxSynth does not save a program if it is recursive or has predicate invention. The reason is that a combination of recursive programs or programs with invented predicates can cover more examples than the union of the examples covered by each individual program. For instance, consider the examples $\{f([1,3]), f([3,0]), f([3,1])\}$ and the hypotheses h_1 and h_2:

$$h_1 = \left\{\ \textit{f(A)} \leftarrow \textit{head(A,1)}\ \right\} \quad h_2 = \left\{ \begin{array}{l} \textit{f(A)} \leftarrow \textit{head(A,0)} \\ \textit{f(A)} \leftarrow \textit{tail(A,B),f(B)} \end{array} \right\}$$

The hypothesis h_1 covers the first example and h_2 covers the second example but the hypothesis $h_1 \cup h_2$ covers all three examples. Therefore, in the combine stage, we cannot simply reason about the coverage of a combination of programs using the union of coverage of the individual programs in the combination. However, MaxSynth can learn MDL programs with recursion or predicate invention as they can be output by the generate stage and evaluated (lines 5-9).

Combine In the combine stage, Popper searches for a combination of promising programs that covers as many positive examples as possible and is minimal in size. By contrast, MaxSynth searches for a combination of promising programs with MDL cost (line 16). The initial maximum MDL cost is the number of positive examples which is the cost of the empty hypothesis. If we find a combination in the combine stage, we update the maximum MDL cost (line 20).

We formulate the search for an MDL combination of programs as a MaxSAT problem (Bacchus, Järvisalo, and Martins 2021). In MaxSAT, given a set of hard clauses and a set of soft clauses with an associated weight, the task is to find a truth assignment which satisfies each hard clause and minimises the sum of the weights of falsified soft clauses.

Our MaxSAT encoding is as follows. For each promising program h, we use a variable p_h to indicate whether h is in the combination. For each example $e \in E^+ \cup E^-$, we use a variable c_e to indicate whether the combination covers e. For each positive example $e \in E^+$, we include the hard clause $c_e \rightarrow \bigvee_{B \cup h \models e} p_h$ to ensure that, if the combination covers e, then at least one of the programs in the combination covers e. For each negative example $e \in E^-$, we include the hard clause $\neg c_e \rightarrow \bigwedge_{B \cup h \models e} \neg p_h$ to ensure that, if the combination does not cover e, then none of the programs in the combination covers e. We encode the MDL cost function as follows. For each promising program h we include the soft clause $(\neg p_h)$ with weight $size(h)$. For each positive example $e \in E^+$, we include the soft clause (c_e) with weight 1. For each negative example $e \in E^-$, we include the soft clause $(\neg c_e)$ with weight 1. We use a MaxSAT solver on this encoding. The MaxSAT solver finds an optimal solution which corresponds to a combination of promising programs that minimises the MDL cost function.

Constrain In the constrain stage (line 21), MaxSynth uses our optimally sound constraints (Section 3.4) to prune the hypothesis space. For instance, given a hypothesis h_1, MaxSynth prunes all generalisations of h_1 with size greater than $|E^+| - fp(h_1)$ (Proposition 2). By contrast, Popper prunes all generalisations of an inconsistent hypothesis.

Correctness We show that MaxSynth returns an optimal noisy solution.

Theorem 1 (**Correctness**)**.** MaxSynth returns an optimal noisy solution if one exists.

Proof. *The proof is in the appendix. We first show that* MaxSynth *without any noisy constraints returns an optimal noisy solution, and then that our noise-tolerant constraints are optimally sound (Propositions 1 and 2).*

5 Experiments

To test our claim that MAXSYNTH can learn programs from noisy data, our experiments aim to answer the question:

Q1 Can MAXSYNTH learn programs from noisy data?

To answer **Q1**, we evaluate MAXSYNTH on a variety of tasks with noisy data. We compare MAXSYNTH against ALEPH (Srinivasan 2001), POPPER, and NOISYPOPPER (Wahlig 2022)[3]. We use these systems because they can learn definite recursive programs. ALEPH is a set covering approach that supports noise. NOISYPOPPER can handle noisy data but can only learn small programs. Because of space limitations and its poor performance, the results for NOISYPOPPER are in the appendix.

To evaluate how MAXSYNTH handles different amounts of noise, our experiments aim to answer the question:

Q2 How well does MAXSYNTH handle progressively more noise?

To answer **Q2**, we evaluate the performance of MAXSYNTH on domains where we can progressively increase the amount of noise. For an increasing noise amount p, we randomly change the label of a proportion p of the training examples.

We claim that our noisy constraints (Section 3.4) can improve learning performance by pruning non-MDL programs from the hypothesis space. To evaluate this claim, our experiments aim to answer the question:

Q3 Can noisy constraints reduce learning times compared to unconstrained learning?

To answer **Q3**, we compare the learning time of MAXSYNTH with and without noisy constraints.

Our approach should improve learning performance when learning programs from noisy data. However, it is often unknown whether the data is noisy. To evaluate the overhead of handling noise, our experiments aim to answer the question:

Q4 What is the overhead of MAXSYNTH on noiseless problems?

To answer **Q4**, we compare the performance of MAXSYNTH and POPPER on standard benchmarks which are not noisy.

Domains We briefly describe our five domains. The appendix includes more details.

IGGP. The goal of *inductive general game playing* (Cropper, Evans, and Law 2020) (IGGP) is to induce rules to explain game traces from the general game playing competition (Genesereth and Björnsson 2013).

Program synthesis. We use a program synthesis dataset (Cropper and Morel 2021). These tasks are list transformation tasks which involve learning recursive programs.

Zendo. Zendo is an inductive game where the goal is to find a rule by building structures of pieces. The game interests cognitive scientists (Bramley et al. 2018).

[3]We considered other systems. Rule selection approaches (Corapi, Russo, and Lupu 2011; Evans and Grefenstette 2018; Kaminski, Eiter, and Inoue 2019; Law 2022) precompute every possible rule which is infeasible on our datasets. Metarule-based approaches (Muggleton, Lin, and Tamaddoni-Nezhad 2015) are unusable in practice (Cropper et al. 2022). Rule learning systems (Galárraga et al. 2015) can only use unary and binary relations.

Alzheimer. These real-world tasks (King, Sternberg, and Srinivasan 1995) involve learning rules describing four properties desirable for drug design against Alzheimer's disease.

Wn18RR. Wn18rr (Bordes et al. 2013) is a real-world knowledge base with 11 relations from WordNet.

Systems MAXSYNTH, POPPER, and NOISYPOPPER use identical biases so the comparison between them is fair. MAXSYNTH uses the UWrMaxSat solver (Piotrów 2020) in the combine stage. To perform a direct comparison, we modify POPPER to also use the UWrMaxSat solver in its combine stage. We use the default cost function (coverage) for ALEPH. We have tried to make a fair comparison with ALEPH but, since it has many additional settings, it is naturally plausible that further parameter tuning could improve its performance (Srinivasan and Ramakrishnan 2011). The appendix contains more details about the systems.

Experimental Setup We measure predictive accuracy (the proportion of correct predictions on unseen test data) and learning time given a maximum learning time of 20 minutes. We repeat all the experiments 10 times and calculate the mean and standard error. We use an 8-Core 3.2 GHz Apple M1 and a single CPU.

5.1 Experimental Results

Experiment 1: Comparison against SOTA Table 1 shows the predictive accuracies of the systems on the datasets. It shows that MAXSYNTH (i) consistently achieves high accuracy on most tasks, and (ii) comprehensively outperforms existing systems in terms of predictive accuracy. A paired t-test shows MAXSYNTH significantly ($p < 0.01$) outperforms POPPER on 25/42 tasks, achieves similar accuracies on 13/42 tasks, and is significantly outperformed by POPPER on 4/42 tasks. For instance, MAXSYNTH has high accuracy (at least 94%) on all *zendo* tasks while POPPER struggles when there is noise. While POPPER searches for a hypothesis that entails as many positive examples as possible and no negative examples, MAXSYNTH tolerates both misclassified positive and negative examples.

MAXSYNTH outperforms ALEPH on the recursive tasks because ALEPH struggles to learn recursive programs. MAXSYNTH also outperforms ALEPH on some non-recursive tasks. For instance, on *iggp-coins*, MAXSYNTH achieves 100% predictive accuracy on the testing examples even with 20% noise in the training examples. One reason is that ALEPH does not consider the size of a hypothesis in its (default) cost function and thus often overfits. ALEPH also sometimes timeouts on *wn18rr* tasks and does not return any hypothesis.

MAXSYNTH does not always achieve 100% predictive accuracy despite learning an MDL hypothesis, such as on the *iggp-md* tasks. The reason is that an MDL hypothesis is not necessarily the hypothesis with the highest predictive accuracy (Domingos 1999; Zahálka and Zelezný 2011).

MAXSYNTH and POPPER are anytime systems. If the search time exceeds a timeout, MAXSYNTH and POPPER return the best hypothesis found thus far. MAXSYNTH terminates on all *iggp*, *zendo*, and *alzheimer* tasks, which means it learns an MDL solution. MAXSYNTH returns the best solution found within timeout for most *program synthesis* tasks

Task	**MaxSynth**	**Popper**	**Aleph**
iggp-md (0)	75 ± 0	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$
iggp-md (10)	65 ± 4	76 ± 6	$\mathbf{85 \pm 6}$
iggp-md (20)	58 ± 4	68 ± 6	$\mathbf{73 \pm 7}$
iggp-buttons (0)	80 ± 0	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$
iggp-buttons (10)	$\mathbf{79 \pm 1}$	$\mathbf{79 \pm 3}$	50 ± 0
iggp-buttons (20)	$\mathbf{77 \pm 1}$	63 ± 1	50 ± 0
iggp-coins (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	50 ± 0
iggp-coins (10)	$\mathbf{100 \pm 0}$	54 ± 1	50 ± 0
iggp-coins (20)	$\mathbf{100 \pm 0}$	50 ± 0	50 ± 0
iggp-rps (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$
iggp-rps (10)	$\mathbf{100 \pm 0}$	63 ± 2	73 ± 3
iggp-rps (20)	$\mathbf{100 \pm 0}$	59 ± 2	50 ± 0
zendo1 (0)	99 ± 0	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$
zendo1 (10)	$\mathbf{99 \pm 0}$	82 ± 1	82 ± 1
zendo1 (20)	$\mathbf{99 \pm 0}$	74 ± 1	76 ± 1
zendo2 (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$
zendo2 (10)	$\mathbf{100 \pm 0}$	64 ± 1	65 ± 1
zendo2 (20)	$\mathbf{100 \pm 0}$	58 ± 1	60 ± 1
zendo3 (0)	98 ± 0	$\mathbf{99 \pm 0}$	$\mathbf{99 \pm 0}$
zendo3 (10)	$\mathbf{98 \pm 0}$	72 ± 1	72 ± 1
zendo3 (20)	$\mathbf{97 \pm 1}$	68 ± 1	70 ± 1
zendo4 (0)	98 ± 0	$\mathbf{99 \pm 0}$	$\mathbf{99 \pm 0}$
zendo4 (10)	$\mathbf{96 \pm 0}$	81 ± 1	82 ± 1
zendo4 (20)	$\mathbf{94 \pm 1}$	74 ± 1	77 ± 1
dropk (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	55 ± 5
dropk (10)	$\mathbf{100 \pm 0}$	54 ± 1	50 ± 0
dropk (20)	$\mathbf{100 \pm 0}$	54 ± 1	50 ± 0
evens (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	57 ± 3
evens (10)	$\mathbf{100 \pm 0}$	52 ± 1	52 ± 1
evens (20)	$\mathbf{100 \pm 0}$	51 ± 0	51 ± 0
reverse (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	50 ± 0
reverse (10)	$\mathbf{100 \pm 0}$	52 ± 0	50 ± 0
reverse (20)	$\mathbf{100 \pm 0}$	52 ± 0	50 ± 0
sorted (0)	$\mathbf{100 \pm 0}$	$\mathbf{100 \pm 0}$	68 ± 5
sorted (10)	$\mathbf{100 \pm 0}$	64 ± 2	63 ± 2
sorted (20)	$\mathbf{100 \pm 0}$	58 ± 1	56 ± 2
alzheimer_acetyl	$\mathbf{68 \pm 1}$	56 ± 0	50 ± 0
alzheimer_amine	$\mathbf{76 \pm 1}$	69 ± 1	73 ± 1
alzheimer_mem	$\mathbf{63 \pm 1}$	51 ± 0	61 ± 1
alzheimer_toxic	74 ± 1	64 ± 1	$\mathbf{83 \pm 1}$
wn18rr1	$\mathbf{98 \pm 0}$	95 ± 1	50 ± 0
wn18rr2	$\mathbf{79 \pm 1}$	78 ± 1	50 ± 0

Table 1: Predictive accuracies. Numbers in parentheses indicate the amount of noise added. The amount of noise is unknown when unspecified.

and *wn18rr2*. To understand how accuracy varies with learning time, we set a timeout of t seconds for increasing values of t. Figure 1 shows the accuracies of the best hypothesis found when increasing the timeout. This result shows that MaxSynth can often quickly find an optimal (MDL) hypothesis. For instance, on *alzheimer-toxic*, MaxSynth takes only 13s to find an optimal hypothesis but needs 48s to prove that this hypothesis is optimal. Likewise, on *zendo2 (20)*, MaxSynth takes only 60s to find an optimal hypothesis but needs 151s more to prove this hypothesis is optimal.

Overall, these results suggest that the answer to **Q1** is that MaxSynth can (i) learn programs, including recursive programs, with high accuracy from noisy data, and (ii) outperform existing systems in terms of predictive accuracies.

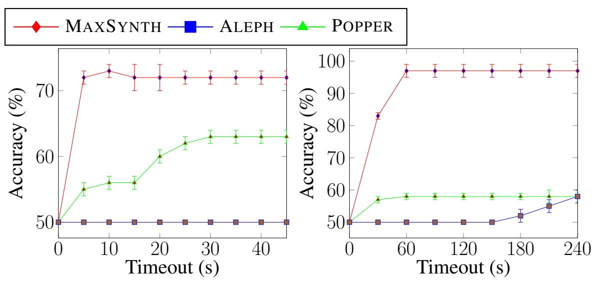

Figure 1: Predictive accuracy with larger timeouts on *alzheimer_toxic* (left) and *zendo2 (20)* (right).

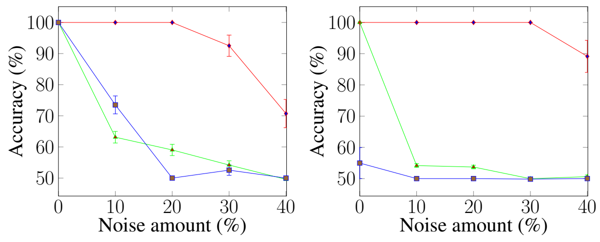

Figure 2: Predictive accuracy versus the noise amount on *iggp-rps* (left) and *dropk* (right).

Experiment 2: Noise Tolerance Figure 2 shows the predictive accuracies of the systems on two tasks when increasing the amount of noise. These results show that the performance of MaxSynth degrades slower with increasing amounts of noise compared to Popper. MaxSynth can scale to problems with up to 30% of noise while Popper struggles from 10% of noise. For instance, on *iggp-rps* with 30% of noise, Popper and Aleph have less than 55% accuracy, whereas MaxSynth has over 90% accuracy. Popper is not robust to false negatives. It returns a hypothesis which is consistent but may only cover a fraction of the positive examples. Aleph typically overfits the data. Overall, these results suggest that the answer to **Q2** is that MaxSynth can scale to moderate amounts of noise.

Experiment 3: Noisy Constraints Table 2 shows the learning times of MaxSynth with and without noisy constraints (Section 3.4). It shows that our constraints can drastically reduce learning times. A paired t-test confirms the significance of the difference for all tasks ($p < 0.01$). The appendix shows the predictive accuracies, which are equal or higher with noisy constraints. This result shows that our noisy constraints are highly effective at soundly pruning the hypothesis space. For instance, on *iggp-md (10)*, MaxSynth considers 21,025 programs without constraints and only 136 programs with constraints, a 99% reduction. Similarly, MaxSynth considers 176,453 programs for *zendo2 (20)* without constraints and only 5,503 programs with constraints, a 97% reduction. The overhead of analysing hypotheses and imposing constraints is small. For instance, on *iggp-md (10)*, MaxSynth spends 0.7s building constraints but this pruning reduces the total

Task	**Without**	**With**	**Difference**
iggp-md (0)	14 ± 0	1 ± 0	**-92%**
iggp-md (10)	109 ± 2	2 ± 0	**-98%**
iggp-md (20)	103 ± 1	2 ± 0	**-98%**
iggp-buttons (0)	61 ± 0	7 ± 0	**-88%**
iggp-buttons (10)	61 ± 1	9 ± 0	**-85%**
iggp-buttons (20)	57 ± 0	10 ± 0	**-82%**
iggp-coins (0)	615 ± 5	138 ± 1	**-77%**
iggp-coins (10)	631 ± 14	141 ± 2	**-77%**
iggp-coins (20)	596 ± 2	144 ± 2	**-75%**
iggp-rps (0)	195 ± 1	50 ± 1	**-74%**
iggp-rps (10)	197 ± 1	60 ± 2	**-69%**
iggp-rps (20)	193 ± 1	66 ± 1	**-65%**
zendo1 (0)	33 ± 9	13 ± 3	**-60%**
zendo1 (10)	648 ± 3	77 ± 4	**-88%**
zendo1 (20)	688 ± 7	100 ± 13	**-85%**
zendo2 (0)	603 ± 4	48 ± 1	**-92%**
zendo2 (10)	611 ± 1	48 ± 3	**-92%**
zendo2 (20)	766 ± 70	118 ± 36	**-84%**
zendo3 (0)	613 ± 2	49 ± 3	**-92%**
zendo3 (10)	626 ± 2	62 ± 2	**-90%**
zendo3 (20)	834 ± 66	190 ± 112	**-77%**
zendo4 (0)	594 ± 4	43 ± 3	**-92%**
zendo4 (10)	616 ± 3	58 ± 2	**-90%**
zendo4 (20)	767 ± 36	122 ± 32	**-84%**
dropk (0)	541 ± 5	7 ± 1	**-98%**
evens (0)	770 ± 2	7 ± 0	**-99%**
reverse (0)	*timeout*	45 ± 7	**-96%**
sorted (0)	1182 ± 8	31 ± 3	**-97%**
alzheimer_acetyl	*timeout*	133 ± 5	**-88%**
alzheimer_amine	*timeout*	73 ± 3	**-93%**
alzheimer_mem	*timeout*	79 ± 3	**-93%**
alzheimer_toxic	*timeout*	61 ± 6	**-94%**
wn18rr1	*timeout*	534 ± 14	**-55%**

Table 2: Learning time for MAXSYNTH with and without noisy constraints. We show tasks where approaches differ. The full table is in the appendix.

learning time from 109s to 2s. Overall, these results suggest that the answer to **Q3** is that noisy constraints can drastically reduce learning times.

Experiment 4: Overhead Table 1 shows the predictive accuracies of MAXSYNTH and POPPER on noiseless problems. These results show that MAXSYNTH often can find a non-noisy solution. However, MAXSYNTH may return a simpler hypothesis than POPPER. For instance, on *iggp-md (0)*, MAXSYNTH returns a hypothesis of size 5. This hypothesis misclassifies 2 training positive examples and therefore has a cost of 7. Its predictive accuracy is 75%. By contrast, POPPER finds a hypothesis of size 11 with maximal predictive accuracy (100%) on the test data. As Vitányi and Li (2000) discuss, MDL interprets perfect data as data obtained from a simpler hypothesis subject to measuring errors.

Table 3 shows the learning times. It shows that MAXSYNTH often has similar learning times to POPPER. For instance, both systems require 7s on *iggp-buttons (0)* and around 50s on *iggp-rps (0)*. MAXSYNTH can be faster than

Task	**MAXSYNTH**	**POPPER**
iggp-md (0)	$\mathbf{1 \pm 0}$	11 ± 0
iggp-buttons (0)	$\mathbf{7 \pm 0}$	$\mathbf{7 \pm 0}$
iggp-coins (0)	$\mathbf{138 \pm 1}$	147 ± 2
iggp-rps (0)	$\mathbf{50 \pm 1}$	53 ± 1
zendo1 (0)	$\mathbf{13 \pm 3}$	23 ± 4
zendo2 (0)	$\mathbf{48 \pm 1}$	102 ± 2
zendo3 (0)	$\mathbf{49 \pm 3}$	85 ± 3
zendo4 (0)	$\mathbf{43 \pm 3}$	79 ± 4
dropk (0)	7 ± 1	$\mathbf{4 \pm 1}$
evens (0)	$\mathbf{7 \pm 0}$	8 ± 0
reverse (0)	$\mathbf{45 \pm 7}$	65 ± 9
sorted (0)	31 ± 3	$\mathbf{19 \pm 2}$

Table 3: Learning times on non-noisy tasks.

POPPER. A paired t-test shows MAXSYNTH significantly outperforms POPPER on 7/12 tasks ($p < 0.01$). For instance, on *zendo2 (0)*, MAXSYNTH takes 48s whilst POPPER takes 102s. The pruning by MAXSYNTH can be effective. For instance, given any hypothesis h, MAXSYNTH prunes specialisations of size greater than $fp(h)$, whereas POPPER only prunes specialisations of consistent hypotheses. Also, MAXSYNTH sometimes returns a smaller hypothesis than POPPER and thus searches up to a smaller depth, such as for *iggp-md (0)*.

Overall, these results suggest that the answer to **Q4** is that unnecessarily tolerating noise is not prohibitively expensive and often leads to similar performance.

6 Conclusions and Limitations

We have introduced an ILP approach that learns MDL programs from noisy examples, including recursive programs. Our approach first learns small programs that generalise a subset of the positive examples and then combines them to build an MDL program. We implemented our idea in MAXSYNTH, which uses a MaxSAT solver to find an MDL combination of programs. Our empirical results on multiple domains show that MAXSYNTH can (i) substantially improve predictive accuracies compared to other systems, and (ii) scale to moderate amounts of noise (30%). Our results also show that our noisy constraints can reduce learning times by 99%. Overall, this paper shows that MAXSYNTH can learn accurate hypotheses for noisy problems that other systems cannot.

Limitations. We use MDL as our criterion for optimality. Our experiments show that an MDL hypothesis does not necessarily have the lowest generalisation error, as discussed by Domingos (1999). To overcome this limitation, future work should investigate alternative cost functions (Lavrac, Flach, and Zupan 1999). For instance, Hernández-Orallo and García-Varea (2000) discuss creative alternatives to MDL.

7 Appendices, Code, and Data

A longer version of this paper with the appendices is available at https://arxiv.org/pdf/2308.09393.pdf. The experimental code and data are available at https://github.com/celinehocquette/aaai24-maxsynth.

Acknowledgements

The first and fourth authors are supported by the EPSRC fellowship (EP/V040340/1). The second and third authors are supported by the Research Council of Finland (grants 347588 and 356046). For open access, the authors have applied a CC BY public copyright licence to any author-accepted manuscript version arising from this submission. The authors thank David Cerna, Sebastijan Dumančić, and Oghenejokpeme Orhobor for valuable feedback.

References

Ahlgren, J.; and Yuen, S. Y. 2013. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Machine Learning Res.*, 14(1): 3649–3682.

Bacchus, F.; Järvisalo, M.; and Martins, R. 2021. Maximum Satisfiabiliy. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 929–991. IOS Press.

Bembenek, A.; Greenberg, M.; and Chong, S. 2023. From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems. *Proc. ACM Program. Lang.*, 7(POPL): 185–217.

Blockeel, H.; and De Raedt, L. 1998. Top-Down Induction of First-Order Logical Decision Trees. *Artif. Intell.*, 101(1-2): 285–297.

Bordes, A.; Usunier, N.; García-Durán, A.; Weston, J.; and Yakhnenko, O. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, 2787–2795.

Bramley, N.; Rothe, A.; Tenenbaum, J.; Xu, F.; and Gureckis, T. M. 2018. Grounding Compositional Hypothesis Generation in Specific Instances. In *Proceedings of the 40th Annual Meeting of the Cognitive Science Society, CogSci 2018*.

Conklin, D.; and Witten, I. H. 1994. Complexity–Based Induction. *Machine Learning*, 16: 203–225.

Corapi, D.; Russo, A.; and Lupu, E. 2011. Inductive Logic Programming in Answer Set Programming. In *Inductive Logic Programming - 21st International Conference*, volume 7207, 91–97.

Cropper, A.; and Dumancic, S. 2022. Inductive Logic Programming At 30: A New Introduction. *J. Artif. Intell. Res.*, 74: 765–850.

Cropper, A.; Dumancic, S.; Evans, R.; and Muggleton, S. H. 2022. Inductive logic programming at 30. *Mach. Learn.*, 111(1): 147–172.

Cropper, A.; Evans, R.; and Law, M. 2020. Inductive general game playing. *Mach. Learn.*, 109(7): 1393–1434.

Cropper, A.; and Hocquette, C. 2023. Learning Logic Programs by Combining Programs. In *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372, 501–508. IOS Press.

Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Mach. Learn.*, 110(4): 801–856.

Cropper, A.; and Muggleton, S. H. 2019. Learning efficient logic programs. *Mach. Learn.*, 108(7): 1063–1083.

Cucala, D. J. T.; Grau, B. C.; and Motik, B. 2022. Faithful Approaches to Rule Learning. In Kern-Isberner, G.; Lakemeyer, G.; and Meyer, T., eds., *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022, Haifa, Israel. July 31 - August 5, 2022*.

Dai, W.; and Muggleton, S. H. 2021. Abductive Knowledge Induction from Raw Data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 1845–1851. ijcai.org.

De Raedt, L.; Dries, A.; Thon, I.; den Broeck, G. V.; and Verbeke, M. 2015. Inducing Probabilistic Relational Rules from Probabilistic Examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1835–1843.

Domingos, P. M. 1999. The Role of Occam's Razor in Knowledge Discovery. *Data Min. Knowl. Discov.*, 3(4): 409–425.

Evans, R.; and Grefenstette, E. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Intell. Res.*, 61: 1–64.

Evans, R.; Hernández-Orallo, J.; Welbl, J.; Kohli, P.; and Sergot, M. J. 2021. Making sense of sensory input. *Artif. Intell.*, 293: 103438.

Galárraga, L.; Teflioudi, C.; Hose, K.; and Suchanek, F. M. 2015. Fast rule mining in ontological knowledge bases with AMIE+. *VLDB J.*, 24(6): 707–730.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694.

Genesereth, M. R.; and Björnsson, Y. 2013. The International General Game Playing Competition. *AI Magazine*, 34(2): 107–111.

Hernández-Orallo, J.; and García-Varea, I. 2000. Explanatory and creative alternatives to the MDL priciple. *Foundations of Science*, 5: 185–207.

Huang, J.; and Pearce, A. R. 2007. Collaborative Inductive Logic Programming for Path Planning. In Veloso, M. M., ed., *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 1327–1332.

Jain, A.; Gautrais, C.; Kimmig, A.; and Raedt, L. D. 2021. Learning CNF Theories Using MDL and Predicate Invention. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*, 2599–2605. ijcai.org.

Kaminski, T.; Eiter, T.; and Inoue, K. 2019. Meta-Interpretive Learning Using HEX-Programs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6186–6190.

King, R. D.; Sternberg, M. J.; and Srinivasan, A. 1995. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing*, 13: 411–433.

Lavrac, N.; Flach, P. A.; and Zupan, B. 1999. Rule Evaluation Measures: A Unifying View. In Dzeroski, S.; and Flach, P. A., eds., *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*, volume 1634 of *Lecture Notes in Computer Science*, 174–185. Springer.

Law, M. 2022. Conflict-driven Inductive Logic Programming. arXiv:2101.00058.

Lin, D.; Dechter, E.; Ellis, K.; Tenenbaum, J. B.; and Muggleton, S. 2014. Bias reformulation for one-shot function induction. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263, 525–530.

Lloyd, J. W. 2012. *Foundations of logic programming*. Springer Science & Business Media.

McCreath, E.; and Sharma, A. 1997. ILP with Noise and Fixed Example Size: A Bayesian Approach. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, 1310–1315. Morgan Kaufmann.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing*, 8(4): 295–318.

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Comput.*, 13(3&4): 245–286.

Muggleton, S.; Dai, W.; Sammut, C.; Tamaddoni-Nezhad, A.; Wen, J.; and Zhou, Z. 2018. Meta-Interpretive Learning from noisy images. *Mach. Learn.*, 107(7): 1097–1118.

Muggleton, S.; and De Raedt, L. 1994. Inductive Logic Programming: Theory and Methods. *J. Log. Program.*, 19/20: 629–679.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Mach. Learn.*, 100(1): 49–73.

Muggleton, S. H.; Srinivasan, A.; and Bain, M. 1992. Compression, Significance, and Accuracy. In Sleeman, D. H.; and Edwards, P., eds., *Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992*, 338–347. Morgan Kaufmann.

Oblak, A.; and Bratko, I. 2010. Learning from Noisy Data Using a Non-covering ILP Algorithm. In Frasconi, P.; and Lisi, F. A., eds., *Inductive Logic Programming - 20th International Conference, ILP 2010, Florence, Italy, June 27-30, 2010. Revised Papers*, volume 6489 of *Lecture Notes in Computer Science*, 190–197. Springer.

Piotrów, M. 2020. UWrMaxSat: Efficient Solver for MaxSAT and Pseudo-Boolean Problems. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 132–136.

Purgal, S. J.; Cerna, D. M.; and Kaliszyk, C. 2022. Learning Higher-Order Logic Programs From Failures. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, 2726–2733. ijcai.org.

Quinlan, J. R. 1990. Learning Logical Definitions from Relations. *Mach. Learn.*, 5: 239–266.

Raghothaman, M.; Mendelson, J.; Zhao, D.; Naik, M.; and Scholz, B. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.*, 4(POPL): 62:1–62:27.

Raychev, V.; Bielik, P.; Vechev, M.; and Krause, A. 2016. Learning programs from noisy data. *ACM Sigplan Notices*, 51(1): 761–774.

Rissanen, J. 1978. Modeling by shortest data description. *Autom.*, 14(5): 465–471.

Si, X.; Raghothaman, M.; Heo, K.; and Naik, M. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6117–6124.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.

Srinivasan, A.; and Ramakrishnan, G. 2011. Parameter Screening and Optimisation for ILP using Designed Experiments. *J. Mach. Learn. Res.*, 12: 627–662.

Vitányi, P. M.; and Li, M. 2000. Minimum description length induction, Bayesianism, and Kolmogorov complexity. *IEEE Transactions on information theory*, 46(2): 446–464.

Wahlig, J. 2022. Learning Logic Programs From Noisy Failures. *CoRR*, abs/2201.03702.

Zahálka, J.; and Zelezný, F. 2011. An experimental test of Occam's razor in classification. *Mach. Learn.*, 82(3): 475–481.

Zeng, Q.; Patel, J. M.; and Page, D. 2014. QuickFOIL: Scalable Inductive Logic Programming. *Proc. VLDB Endow.*, 8(3): 197–208.