Bias reformulation for one-shot function induction

Dianhuan Lin[1] and **Eyal Dechter**[1] and **Kevin Ellis**[1] and **Joshua B. Tenenbaum**[1] and **Stephen H. Muggleton**[2]

Abstract. In recent years predicate invention has been underexplored as a bias reformulation mechanism within Inductive Logic Programming due to difficulties in formulating efficient search mechanisms. However, recent papers on a new approach called Meta-Interpretive Learning have demonstrated that both predicate invention and learning recursive predicates can be efficiently implemented for various fragments of definite clause logic using a form of abduction within a meta-interpreter. This paper explores the effect of bias reformulation produced by Meta-Interpretive Learning on a series of Program Induction tasks involving string transformations. These tasks have real-world applications in the use of spreadsheet technology. The existing implementation of program induction in Microsoft's FlashFill (part of Excel 2013) already has strong performance on this problem, and performs one-shot learning, in which a simple transformation program is generated from a single example instance and applied to the remainder of the column in a spreadsheet. However, no existing technique has been demonstrated to improve learning performance over a series of tasks in the way humans do. In this paper we show how a functional variant of the recently developed Metagol$_D$ system can be applied to this task. In experiments we study a regime of layered bias reformulation in which size-bounds of hypotheses are successively relaxed in each layer and learned programs re-use invented predicates from previous layers. Results indicate that this approach leads to consistent speed increases in learning, more compact definitions and consistently higher predictive accuracy over successive layers. Comparison to both FlashFill and human performance indicates that the new system, Metagol$_{DF}$, has performance approaching the skill level of both an existing commercial system and that of humans on one-shot learning over the same tasks. The induced programs are relatively easily read and understood by a human programmer.

1 Introduction

A remarkable aspect of human intelligence is the ability to learn a general principle, concept, or procedure from a single instance. Suppose you were told a computer program outputs "BOB" on input "bob." What will it produce on input "alice"? Will it return "BOB" again, ignoring the input? Or perhaps it will return "BOLICE", postpending to "BO" the all-caps transform of the input minus the first two characters. Is it reasonable to think it returns the all-caps palindrome formed by all but the last letter of the input, so "alice" maps to "ALICILA"? In practice most people will predict the program will return "ALICE", and not any of the above possibilities. Similarly, guessing the program associated with any of the input-output pairs in the rows of Table 1 seems straightforward, but the space of possible consistent transformations is deceptively large. The reason these problems are easy for us, but often difficult for automated systems, is that we bring to bear a wealth of knowledge about which kinds of programs are more or less likely to reflect the intentions of the person who wrote the program or provided the example.

	Input	*Output*
Task1	miKe dwIGHT	Mike Dwight
Task2	European Conference on Artificial Intelligence	ECAI
Task3	My name is John.	John

Figure 1. Input-output pairs typifying string transformations in this paper

There are a number of difficulties associated with successfully completing such a task. One is inherent ambiguity: how should one choose from the vast number of consistent procedures? There is no clear objective function to minimize; nor is this objective function a subjective utility an intelligent agent can set arbitrarily since there is generally a consensus regarding the "right" answer. Another difficulty is these consistent procedures lie sparsely within a vast set of inconsistent procedures: how can one quickly search this space and find a procedure satisfying the given example without directing the search to trivial solutions? Finally, there is the difficulty of inductive programming in general: the space of such procedures is unruly. Syntactically similar procedures are not in general semantically similar. General heuristics do not exist that reliably estimate a distance to a solution from a partial solution or an incorrect solution.

It is often most effective and natural to teach another person a new behavior or idea by providing a few examples, and the ease with which someone extrapolates from a few examples seems to be a hallmark of intelligence in general and of expertise in specific domains. To produce intelligent robots and interactive software that can flexibly engage in novel tasks, we need to understand how such learning can be accomplished. The literature on Programming By Example has explored many of these questions, with the goal of producing end-user software that automates repetitive tasks without requiring a programmer's expertise. For the most part, the tools produced by these systems have not reached levels of accuracy, flexibility, and performance suitable for end-user adoption [9].

Recent work by Gulwani et al [8] demonstrates that a carefully engineered Domain Specific Language (DSL) for string transformations allows their system to induce string transformations from a single input-output example with speeds and accuracies suitable for commercial software. In further work [7], they demonstrate that careful crafting of a DSL results in impressive inductive programming in other domains such as intelligent tutoring.

The research presented here is intended as a first response to the

[1] Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, USA

[2] Department of Computing, Imperial College London, UK, email: s.muggleton@imperial.ac.uk

challenge Gulwani et al's work seems to pose to AI: if carefully crafted DSLs are a key ingredient for competent one-shot induction of programs, then can we develop AI systems that attain such competence by automatically learning these DSLs?

DSLs are useful, both for human and automated programmers, because they provide a correspondence between the semantic abstractions relevant to the domain and the syntactic elements provided by the language. In doing so, they make programs that fall in the target domain easy to express, easy to reason about, and concise. For a search algorithm exploring the space of programs, this amounts to imposing a bias on the search trajectory.

Metagol$_D$ [13] is an Inductive Logic Programming (ILP) system that uses the recently developed Meta-Interpretive Learning framework to induce logical predicates from positive and negative examples. Metagol$_D$ is able to invent intermediate predicates to facilitate the definition of target predicates. For example, asked to learn the concept *ancestor*, and given *father* and *mother* relationships between individuals, Metagol$_D$ will automatically invent the predicate *parent*. Since our goal in this work is one-shot function induction in a multitask setting we use a functional variant of Metagol$_D$ called Metagol$_{DF}$ that uses predicates invented in the service of one task to facilitate solving other, perhaps more difficult, tasks. In this way, we can use Metagol$_{DF}$ as a model of the utility and practicality of learning a DSL that biases a search over programs.

Our contribution In this paper, we introduce Metagol$_{DF}$ and apply it to the domain of string transformations to explore three aspects of multi-task learning and bias reformulation in inductive programming:

- we show incremental predicate invention using a revised version of Metagol can generate a domain-specific bias that improves the speed and performance of one-shot program induction;
- we compare a general purpose inductive learner to humans and FlashFill and compare the degree to which a simple cypher influences their relative performances.
- we show that an ILP system, although primarily used to learn relational predicates, can be used to learn functional predicates.

1.1 Related work

Both the challenge of learning computer programs and of learning an inductive bias from related problems have a long history in AI, Machine Learning, and other fields [19, 2, 16]. A somewhat smaller literature relates these two problems (eg [15]). Work on statistical machine learning and neural networks has studied transfer of knowledge from one problem domain to other problems domains and has empirically explored the utility of such transfer learning (for an overview see [18]). A theoretic model of "learning to learn" is presented in [1].

More recently, multitask learning has been framed as inference in a hierarchical Bayesian model [20, 4]. This framing has been used to apply these ideas to multitask learning of functional programs [5, 11], where a declarative bias is learned via inference over a latent grammar on programs. Liang et al [11] uses a stochastic search over programs and Dechter et al [5] uses an enumeration over programs, and both represent programs in the combinatory calculus. By contrast, Metagol$_{DF}$ represents functions as logic programs and uses SLD-resolution to guide program search. This results in more interpretable programs and a potentially more intelligent search.

The field of Programming By Demonstration (and also known as Programming By Example) aims to create systems that automatically induce computer programs in order to facilitate human-computer interactions [3]. Lau et al [10] applied the Inductive Logic Programming system FOIL to Programming By Demonstration, but Metagol$_{DF}$ is a qualitatively different approach to Inductive Logic Programming which enables predicate invention and thus learning a bias in the multitask setting.

The work in this paper is directly inspired by recent advances in Programming Demonstration which use DSLs for various domains of interest (see [6, 17, 21]). These approaches demonstrate the power of DSLs in enabling efficient and reliable automated programming. At least one attempt [12] has been made to extend this approach by learning feature weights to guide the search but it does not learn new features of program fragments.

2 Meta-Interpretive Learning framework

The framework described in this section is an extension of that found in [13, 14]. The approach is based on an adapted version of a Prolog meta-interpreter. Normally such a meta-interpreter derives a proof by repeatedly fetching first-order Prolog clauses whose heads unify with a given goal. By contrast, a meta-interpretive learner additionally fetches higher-order metarules whose heads unify with the goal, and saves the resulting meta-substitutions to form an hypothesis. To illustrate the idea consider the metarule below.

Name	**Meta-Rule**
Chain	$P(x,y) \leftarrow Q(x,z), R(z,y)$

The uppercase letters P, Q, R denote existentially quantified higher-order variables while the lowercase letters x, y, z are universally quantified first-order variables. In the course of a proof meta-substitutions for the existentially quantified variables are saved in an abduction store. For instance, suppose the higher-order substitution $\theta = \{P/aunt, Q/sister, R/parent\}$ applied to the *Chain* metarule above allows the proof to complete. In this case the higher-order ground atom $chain(aunt, sister, parent)$ is saved in the abduction store. Given this ground atom the substitution θ can be reconstructed and re-used in later proofs, allowing a form of inductive programming which supports both predicate invention and the learning of recursive definitions [13]. Following the proof of a goal consisting of a set of examples, the hypothesised program is formed by applying the resulting meta-substitutions to their corresponding metarules.

Example 1 Meta-substitution example. *If the examples are* $\{aunt(mary, harry),\ aunt(jane, emma)\}$ *and we have background knowledge* $\{sister(mary, lisa), parent(lisa, harry),\ sister(jane, jack), parent(jack, emma)\}$ *then abducing the statement* $chain(aunt, sister, parent)$*, representing the meta-substitution* θ *above, results in the hypothesised clause* $aunt(x,y) \leftarrow sister(x,z), parent(z,y)$.

2.1 Language classes, expressivity and complexity

The metarules can be viewed as limiting the hypothesis space to being within a particular language class. For instance, the Chain rule above resticts hypothesised clauses to be definite with two atoms in the body and having predicates of arity two. This restriction represents a subset of the language class H^2_2, which includes all datalog definite logic programs with at most two atoms in the body of each clause and having predicates of arity at most two.

Theorem 1 *(The number of H^2_2 programs of size n.) Given p predicate symbols and m metarules the number of H^2_2 programs expressible with n clauses is $O(m^n p^{3n})$.*

Proof. *The number of clauses S_p which can be constructed from an H^2_2 metarule given p predicate symbols is at most p^3. Therefore the set of such clauses $S_{m,p}$ which can be constructed from m*

Generalised meta-interpreter
$prove([], Prog, Prog).$
$prove([Atom\|As], Prog1, Prog2) :-$
$\quad metarule(Name, MetaSub, (Atom$:- $Body), Order),$
$\quad Order,$
$\quad abduce(metasub(Name, MetaSub), Prog1, Prog3),$
$\quad prove(Body, Prog3, Prog4),$
$\quad prove(As, Prog4, Prog2).$

Figure 2. Prolog code for generalised meta-interpreter used in Metagol$_{DF}$

Name	Meta-Rule	Order
Base	$P(x,y) \leftarrow Q(x,y)$	$P \succ Q$
Chain	$P(x,y) \leftarrow Q(x,z), R(z,y)$	$P \succ Q, P \succ R$
TailRec	$P(x,y) \leftarrow Q(x,z), P(z,y)$	$P \succ Q,$ $x \succ z \succ y$

Figure 3. Table of Meta-rules with associated Herbrand ordering constraints. $\succ$ is a pre-defined ordering over symbols in the signature.

distinct H^2_2 metarules using p predicate symbols has cardinality at most mp^3. From this it follows that the number of logic programs constructed from a selection of n rules chosen from $S_{m,p}$ is at most $\binom{mp^3}{n} \leq (mp^3)^n = O(m^n p^{3n})$.

Given this exponential growth in the hypothesis space, our implementation (see Section 3) places a bound $n = k$ on the maximum number of clauses in any learned string transformation function.

2.2 String transformation functions

This paper studies the use of Meta-Interpretive Learning for inducing a set of related string transformation functions each having the form $f : \Sigma^* \rightarrow \Sigma^*$ where Σ^* is the set of sequences over a finite alphabet Σ. In order to learn such functions, each dyadic predicate $P(x, y)$ used in the metarules is treated as a function from x to y. Additionally both x and y are treated as $Input/Output$ pairs where $Input$ and $Output$ are sequences from Σ^*.

3 Implementation

This section describes Metagol$_{DF}$, a variant of Metagol$_D$ [13], aimed at learning functions rather than relations.

3.1 Metagol$_{DF}$

Figure 2 shows the implementation of Metagol$_{DF}$[3] as a generalised Meta-Interpreter, similar in form to a standard Prolog meta-interpreter. The meta-rule base (see Figure 3) is defined separately, with each rule having an associated name ($Name$), quantification ($MetaSub$), rule form ($Atom$: $-Body$) and Herbrand ordering constraint ($Order$). This contrasts with Metagol$_D$ [13] in which the meta-rules are not separable from the meta-interpreter. Separating the meta-rules from the meta-interpreter makes it easier for users to define meta-rules. The restriction of relations to functions is implemented as a post-construction test which rejects every hypothesised function R for which there is a positive example $R(x, y)$, while $R(x, z)$ is provable from the hypothesis where $x \neq z$. In practice the functional restriction largely obviates the need for negative examples.

3.2 Herbrand ordering constraints

Owing to the Turing-expressivity of H^2_2 it is necessary [13] to use constraints on the application of the metarules to guarantee termination of the hypothesised program. The termination guarantees are based on these constraints being consistent with a total ordering over the Herbrand base of the hypothesised program. Thus the constraints ensure that the head of each clause is proved on the basis of instances of body atoms lower in the ordering over the Herbrand base. Since the ordering is not infinitely descending, this guarantees termination of the meta-interperpreter. Figure 3 shows the metarules used in Metagol$_{DF}$ alongside their associated constraints.

3.3 Dependent learning

The implementation of the earlier Metagol$_D$ [13] system uses iterative deepening of derivations of the meta-interpreter for each learning episode up to a bound which is logarithmic in the number of examples. This leads to efficiency in the case of large numbers of examples. In this paper we consider the case of learning multiple learning tasks each from a single training example. Since $\log 1 = 0$ we use an alternative approach in which iterative deepening is multiplexed across the set of all learning tasks up to a given maximum depth k. In the remainder of this paper we use the term *Dependent learning* to refer to this approach. Thus Metagol$_{DF}$ starts by setting the depth bound d to 1 and finds all task definitions which can be expressed using a single clause. Next $d = 2$ is tried for all remaining tasks, where each task is allowed to re-use invented predicates from the previous depth bound. The search continues in this way until $d = k$ and returns the set of all learned definitions. Given Theorem 1 the value of k is restricted to 5 in our experiments to avoid excessive runtimes.

3.4 Predicate invention

At each depth d the dependent learning allows the introduction of up to $d - 1$ new predicate symbols. New predicates names are formed by taking the name of the task (say *f4*) and adding underscores and numbers (say *f4_1*, *f4_2*, etc). New predicate symbols are added into the ordering $\succ$ over the signature (see Figure 3) and placed immediately below the name of the task being learned and immediately above the names of all other predicate symbols.

4 Experiments

In this section, we evaluate the performance of Metagol$_{DF}$ on one-shot learning when given different strength of learning biases. We also compare the performance of Metagol$_{DF}$ to Excel's FlashFill and human beings via Mechanical Turk experiments.

4.1 Materials

Metagol$_{DF}$ In order to obtain string transformation tasks corresponding to those naturally found in real-world settings, Gulwani et al [6] compiled a set of typical string transformations from online Microsoft Excel user forums. Since their data is not yet publicly available, we compiled a set of examples from their papers, supplementing these with handcrafted examples in the spirit of common spreadsheet manipulation tasks. This resulted in 30 problems, each with five input-output examples; for an example of five such example pairs, see Figure 4. Out of these 30 problems, there were 17 we judged to be learnable given the primitive functions being considered in this paper. All comparisons in this paper are based on these 17 problems, and we are keen to address the remaining problems in further extensions of this work (see Section 5).

[3] Full code for Metagol$_{DF}$ together with all materials for the experiments in Section 4 can be found at http://ilp.doc.ic.ac.uk/metagolDF/.

miKe dwIGHT	⇒	Mike Dwight
IaN RoDny	⇒	Ian Rodny
StaNleY TRAVis	⇒	Stanley Travis
MELVIN Julian	⇒	Melvin Julian
mary gelman	⇒	Mary Gelman

Figure 4. A string transformation task

```
copyalphanum/2,copy1/2,
write1/3,skip1/2,
skipalphanum/2, skiprest/2,
make_uppercase1/2 .
make_lowercase/2,
make_uppercase/2
```

Figure 5. Primitive operations given to Metagol$_{DF}$

We provide initial background knowledge for Metagol$_{DF}$, by specifying a set of primitive string transformation operations, as shown in Figure 5. Some of these operations only act on a single character in the input or istring. For example, the predicate $make\_lowercase1/2$ reads in the first letter on the input, if it is alphabetical, and writes the lowercase verison of that letter to the output stream. We also define operations that consume and alter strings of multiple consecutive characters. For example, $make\_lowercase/2$, which is written in terms of $make\_lowercase1/2$, (see Figure 6), reads in the longest consecutive string of alphabetical characters and writes its lowercase version to the output string.

We also considered two different sets of metarules: *Non-recursive* based on only the Chain metarule and *Recursive* based on the Chain, Base and TailRec metarules (see Section 2). Clearly, the language generated by the Non-recursive set of metarules is more limited than that generated by the Recursive set of metarules.

Humans and FlashFill To attain points of comparison for the performance of Metagol$_{DF}$, we assessed human subjects and FlashFill on the seventeen problems on which we tested Metagol$_{DF}$. People, and, to a lesser extent, FlashFill, bring to these problems a large amount of background knowledge. In an attempt to understand the impact of such background knowledge on performance, we tested both people and FlashFill on the original set of input-output examples and on a cyphered version. We used a substitution cypher on the string characters that maintains the character category memberships given to Metagol$_{DF}$. Alphabetical characters were mapped to alphabetical characters but upper- and lowercase variants were preserved (i.e. if 'a'→ 'c' then 'A'→ 'C'). Digits were mapped to digits. The remaining symbols, including space, were mapped among themselves. For example, the input-output pair ⟨"2007 (September)", "September"⟩ was mapped to ⟨"8337}Ivanvqwvś'', "Ivanvqwvs"⟩.

4.2 Method

Metagol$_{DF}$ We compare dependent learning to independent learning in terms of predictive accuracy and running time. Considering only one example is used for training while there are five examples in total, leave-four-out (keep-one-in) cross validation is conducted by measuring the predictive accuracy from each example against the remaining four examples, and averaging the result. In the case of dependent learning, different combinations of examples from each task will affect the learning results, therefore we randomly permuted the order of examples within each task, as well as the order of tasks. Then during the leave-four-out cross validation, examples from the same index of each task are drawn to form a sequence of training examples. For example, at the first round, all the first examples from each task are gathered for training, then similarly for the other four rounds. For each task, Metagol$_{DF}$ is given maximum of ten minutes to solve the problem. The average time taken for learning is around one minute. If the time-out bound is reached, it moves to the next task. All the experiments were run on a 1.6 GHz desktop computer with 16 GB of memory available.

```
make_lowercase(X,Y) :- not_alphanum(X).
make_lowercase(X,Y) :- make_lowercase1(X,Z),
                       make_lowercase(Z,Y).
make_lowercase1(X,Y) :- uppercase(X), downcase(X,Y).
make_lowercase1([H|T1]/[H|T2],T1/T2) :- lowercase1(H).
```

Figure 6. Background knowledge (partial)

Humans and FlashFill FlashFill was assessed using the built-in implementation shipped with Microsoft Office 2013. We employed the same evaluation procedure as that in the Metagol$_{DF}$ experiment, that is, leave-four-out cross validation. However, different from Metagol$_{DF}$, the example ordering no longer matters since FlashFill solves each problem independently.

169 human subjects volunteered to do the experiments on Amazon Mechanical Turk (http://www.mturk.org) and each subject was paid $1.00 to provide responses on ten randomly selected responses from the seventeen problems. Half the subjects saw only uncyphered text and half the subjects saw only cyphered text. Each subject was shown one randomly chosen example pair as the training example for a question and was tested on two randomly chosen example pairs. Accuracies were averaged across all questions and participants.

4.3 Results and discussion

Programs derived by dependent learning Figure 7(a) shows a calling diagram for programs derived by dependent learning when learning from the recursive metarules. Examples of a chain of learned definitions with dependencies based on calling other definitions is exemplified in Figure 8. It would be infeasible for such a program to be learned from scratch all at once given the exponential nature of the search. However, the layers of dependencies found by Metagol$_{DF}$ facilitate this form of learning, allowing knowledge to be efficiently and compactly learned in a bottom-up fashion with invented sub-predicates being multiply re-used.

In contrast to Figure 7(a), Figure 7(b) shows the result of independent learning, which exhibits no dependency among the hypothesised programs. Compared to dependent learning, an independent learner has to solve problems at larger size bound due to constructing sub-functions which are not available for re-using. As shown in Figure 7(b), there are more nodes at size bound 5 and time out region. Although task 3 appears at level 5 in both cases, it has only size one in the case of dependent learning due to re-using the functions f_{12} and f_{12_1} derived earlier. When solving the same task with independent learning, the five clauses need to be built entirely from the initial primitive set. Due to the dependency among programs, those

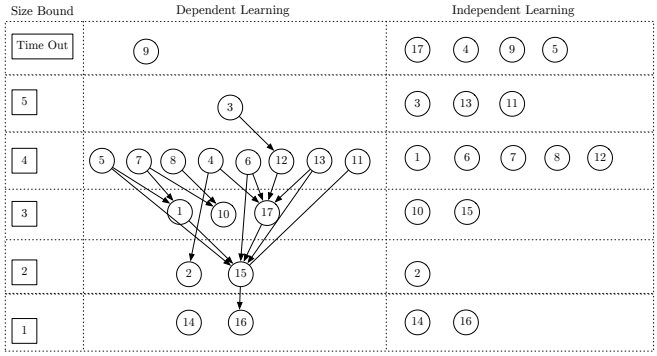

Figure 7. A comparison of the programs generated by Dependent and Independent Learning in a typical run of Metagol$_{DF}$. Nodes marked n correspond to programs which solve task n, and nodes are arranged vertically according to their sizes. For Dependent Learning (left), the arrows correspond to the calling relationships of the induced programs.

derived by dependent learning are more compact than those derived by independent learning.

f_{03}(A,B) :- f_{12_1}(A,C), f_{12}(C,B).

f_{12}(A,B) :- f_{12_1}(A,C), f_{12_2}(C,B).
f_{12_1}(A,B) :- f_{12_2}(A,C), $skip1$(C,B).
f_{12_2}(A,B) :- f_{12_3}(A,C), $write1$(C,B,'.').
f_{12_3}(A,B) :- $copy1$(A,C), f_{17_1}(C,B).

f_{17}(A,B) :- f_{17_1}(A,C), f_{15}(C,B).
f_{17_1}(A,B) :- f_{15_1}(A,C), f_{17_1}(C,B).
f_{17_1}(A,B) :- $skipalphanum$(A,B).

f_{15}(A,B) :- f_{15_1}(A,C), f_{16}(C,B).
f_{15_1}(A,B) :- $skipalphanum$(A,C), $skip1$(C,B).

f_{16}(A,B) :- $copyalphanum$(A,C), $skiprest$(C,B).

Figure 8. Example of a chain of functional logic programs derived by Metagol$_{DF}$ using dependent learning. Note that the hypothesised function f_{03} calls f_{12} which calls f_{17_1}. In turn f_{17} calls f_{15} which calls f_{16}.

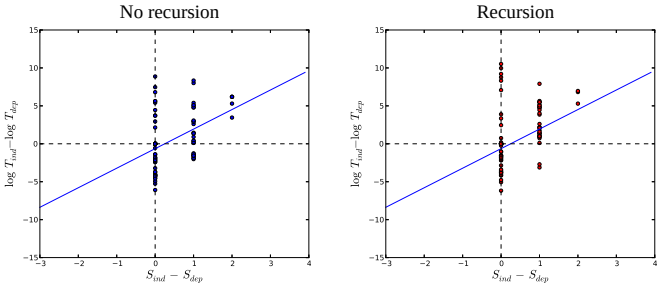

Figure 9. Independent vs. dependent learning: Running time correlated with the size of hypothesised programs

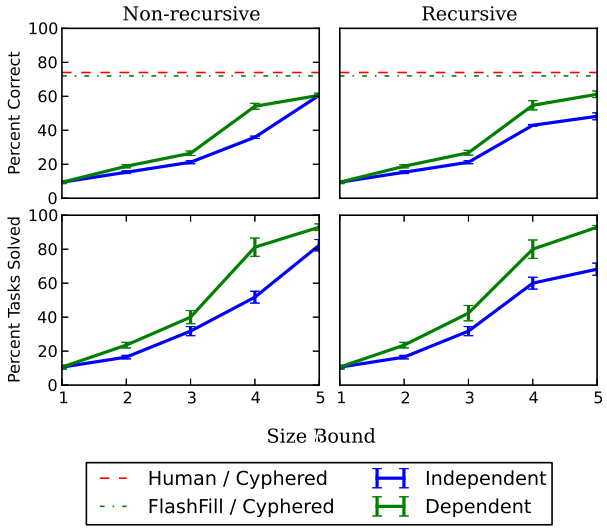

Figure 10. Independent vs. dependent learning: a) predictive accuracy and b) percentage of solved tasks

Independent learning vs. Dependent learning Each graph in Figure 9 depicts the results of five train-and-test runs. Since each point corresponds to learning a problem there are 85 points in total.

The horizontal axis represents the difference between the size of programs derived by independent and dependent learning. All the points are distributed on the positive side of the horizontal axis, which means dependent learning always derives hypotheses with smaller sizes than independent learning. The vertical axis of Figure 9 corresponds to the difference of log running times for independent and dependent learning. Therefore, points distributed above the horizontal axis corresponds to the cases when dependent learning is faster.

According to Theorem 1, the hypothesis space grows exponentially with the size of a program being searched for. Therefore dependent learning's gain in program compactness leads to exponential reducton in the running time. The linear regression line in Figure 9 is consistent with this theoretical result: the gain in speed correlates with the gain in compactness. Independent learning is only faster when there is no size difference or the difference is small, as shown in Figure 9 where the points distributed below horizontal axis gather at $S_{Ind} - S_{Dep} = 0$. This is due to the overhead in dependent learning, since it has a larger number of predicates accumulated from previous depth bounds.

Higher predictive accuracy As shown in Figure 10(a), dependent learning makes it possible to solve more tasks than the independent learning when given the same clause bound. This results in consistently higher predictive accuracies when the learning involves recursive programs. While in the case of learning non-recursive programs, dependent learning still has significantly higher accuracies than independent ones with the exception of size bound 5 where the two accuracy lines converge. The reasons for convergence are: (1) the primitives given to Metagol are strong enough to construct programs without re-using functions learned from other tasks; (2) hypothesis space defined by the non-recursive metarules is small enough that independent learning manages to find a program with large size, without running into time-out mode in the case of learning with recursions. Although learning without recursion restricts the expressivity, good approximations to a target recursive program often exist. Therefore their predictive accuracies are not significantly hampered, only decreasing slightly from 61.2% to 60.6%. However, this is not al-

ways the case, especially when the initial bias is weaker. Then there is higher demand of learning recursions, such as reconstructing the recursive function make_lowercase/2 given in the current initial bias.

In separate trials we investigate such weaker initial bias, consisting of only predicates which altered single characters. It was found that Metagol is able to reformulate recursive functions like make_lowercase/2, producing a monontonic rise in predictive accuracy to a level of around 40%. Notably the starting point of the rise for the weaker bias was delayed owing to the fact the initial concepts to be learned required larger definitions.

Comparison to FlashFill and human performance Figure 10 includes the performance of human beings and the Flashfill on our set of 17 tasks when the strings are cyphered. These results indicate that the performance of Metagol$_{DF}$ approaches the level of both an existing commercial system and that of human beings on one-shot learning over these tasks. Note however, that since we chose these problems with the capabilities of the given primitive set for Metagol$_{DF}$ in mind, we cannot make general claims about the performance of Metagol$_{DF}$ as compared to FlashFill for a wider set of tasks.

For both people and FlashFill, we also acquired performance data for the original uncyphered version of the tasks. The background knowledge we gave to Metagol$_{DF}$ contained no primitives that discriminate between the cyphered and uncyphered tasks, so the performance of Metagol$_{DF}$ is invariant to which version of the tasks were used. By contrast, the human subjects' performance varied significantly depending on whether the cypher was used or not. On the cyphered version of the tasks, human subjects averaged 87% accuracy. On the uncyphered version, human subjects averaged only 74%. FlashFill was much less sensitive to the cypher. It averaged 76% accuracy for uncyphered tasks and 72% for cyphered tasks.

FlashFill encodes a domain specific language which in some cases produces very unintuitive results. For example, FlashFill makes the following prediction"IaN RoDny $\Rightarrow$ Man Ddny" for the learning task shown in Figure 4. This is due to FlashFill's bias of simply copying the first capital letter from training examples. By contrast, Metagol$_{DF}$ makes correct predictions for this problem.

Human beings also make generalisation errors similar to that of Metagol when given the fifth example of task 10: "mary gelman $\Rightarrow$ Mary Gelman". Specifically, Metagol hypothesises a program which only capitalise the first letter of the word and copies the rest. However, the target program makes all non-first letters lowercase. Most subjects do not over fit on this training example due to our background knowledge. However, one human subject who made similar generalisation errors to Metagol.

5 Conclusion and further work

In this paper, we have presented an approach for automatically learning a domain specific bias in a multitask inductive programming setting. This bias reformulation, we argue, is necessary for an intelligent system that can function in new domains and aspires to the kind of one-shot learning that people commonly exhibit. After all, people are not born with a set of inductive biases, one for every possible domain of expertise they might encounter. Domain specific successes in AI – whether in playing board games, folding laundry, or automating spreadsheet operations – pose a challenge to create domain-general systems that can flexibly acquire the appropriate biases whatever the domain of interest. Our work here is meant as a step in that direction within the framework of inductive programming.

Our work leaves many questions unanswered. Most important is how to manage the complexity created by learned predicates. As noted above, each learned predicate increases the branching factor of the search space, and our algorithm, as described above, maintains every learned predicate. By contrast human beings usually compress the previous learned knowledge by further abstraction. Another potential solution has been investigated by Liang et al [11] and Dechter et al [5] who suggest a method to weight the set of invented predicates. This weighted library of primitives could be used to direct search within Metagol$_{DF}$ and prioritize the use of one invented predicate over another. One future direction would be to incorporate such an inference-based library learning within Metagol$_{DF}$.

Although the design of better intelligent user interfaces is one motivation for our work, much remains to evaluate our approach in the context of a working system for human-computer interaction, where active user input and feedback response plays a significant role.

It also remains to be shown that string transformation examples are generalizable to other one-shot function induction tasks. Although the H^2_2 fragment has been demonstrated to have Turing-expressivity, other one-shot learning tasks need to be demonstrated using the same approach. This paper is an initial study of one particular task in one-shot learning, which we hope to extend in further work.

Another question left for future investigation is that of how to learn "algorithmic" biases. Many domain specific algorithms benefit not only from the bias imposed by the choice of representation but also from special algorithmic properties of that representation. For example, the FlashFill algorithm is very efficient because the DSL it uses supports a data structure for compactly representing exponentially large sets of programs consistent with the data [6]. This suggests an area for future research: should automated systems constrain learned representations to those that support these special properties? How might learning in the space of such representations take place?

In future work we hope to also extend the approach to deal with some of the 30-17=13 problems which could not be handled using the primitive tranformations used in this paper. In particular, we hope to investigate the use of primitives which go beyond tranferring characters from the input to the output while maintaining their order. One way this might be possible is by use of operations which push text onto an internal stack. We would also like to investigate ways in which it might be possible to allow effective learning from a weaker initial bias. This might be possible by limiting the number of re-used predicates based on their potential frequency of use.

ACKNOWLEDGEMENTS

The last author would like to thank the Royal Academy of Engineering and Syngenta for funding his present 5 year Research Chair. He would also like to acknowledge the support of Syngenta in its funding of the University Innovations Centre at Imperial College. Additionally we acknowledge support from the Center for Minds, Brains and Machines (CBMM), funded by NSF STC award CCF-1231216.

REFERENCES

[1] Jonathan Baxter, 'A model of inductive bias learning', *J. Artif. Intell. Res.(JAIR)*, **12**, 149–198, (2000).
[2] Shai Ben-David and Reba Schuller, 'Exploiting task relatedness for multiple task learning', in *Learning Theory and Kernel Machines*, 567–580, Springer, (2003).
[3] Allen Cypher and Daniel Conrad Halbert, *Watch what I do: programming by demonstration*, MIT press, 1993.
[4] Hal Daumé III, 'Bayesian multitask learning with latent hierarchies', in *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pp. 135–142. AUAI Press, (2009).
[5] Eyal Dechter, Jonathan Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum, 'Bootstrap learning via modular concept discovery', in *IJCAI*, (2013).

[6] Sumit Gulwani, 'Automating string processing in spreadsheets using input-output examples', *ACM SIGPLAN Notices*, **46**(1), 317–330, (2011).

[7] Sumit Gulwani, 'Example-based learning in computer-aided stem education', Report, Microsoft Research, Redmond, WA, (2013).

[8] Sumit Gulwani, William R Harris, and Rishabh Singh, 'Spreadsheet data manipulation using examples', *Communications of the ACM*, **55**(8), 97–105, (2012).

[9] Tessa Lau et al., 'Why PBD systems fail: Lessons learned for usable AI', in *CHI Workshop on Usable AI*, (2008).

[10] Tessa A Lau and Daniel S Weld, 'Programming by demonstration: An inductive learning formulation', in *Proceedings of the 4th international conference on Intelligent user interfaces*, pp. 145–152. ACM, (1998).

[11] Percy Liang, Michael I. Jordan, and Dan Klein, 'Learning programs: A hierarchical bayesian approach', in *ICML*, pp. 639–646, (2010).

[12] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai, 'A machine learning framework for programming by example', in *Proceedings of The 30th International Conference on Machine Learning*, pp. 187–195, (2013).

[13] S.H. Muggleton and D. Lin, 'Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited', in *Proceedings of the 23rd International Joint Conference Artificial Intelligence (IJCAI 2013)*, pp. 1551–1557, (2013).

[14] S.H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad, 'Meta-interpretive learning: application to grammatical inference', *Machine Learning*, **94**, 25–49, (2014).

[15] J.R. Quinlan and R.M Cameron-Jones, 'FOIL: a midterm report', in *Proceedings of the 6th European Conference on Machine Learning*, ed., P. Brazdil, volume 667 of *Lecture Notes in Artificial Intelligence*, pp. 3–20. Springer-Verlag, (1993).

[16] Michael T Rosenstein, Zvika Marx, Leslie Pack Kaelbling, and Thomas G Dietterich, 'To transfer or not to transfer', in *NIPS 2005 Workshop on Transfer Learning*, volume 898, (2005).

[17] Rishabh Singh and Sumit Gulwani, 'Learning semantic string transformations from examples', *Proceedings of the VLDB Endowment*, **5**(8), 740–751, (2012).

[18] Sebastian Thrun, 'Learning to learn: Introduction', in *In Learning To Learn*. Citeseer, (1996).

[19] Ricardo Vilalta and Youssef Drissi, 'A perspective view and survey of meta-learning', *Artificial Intelligence Review*, **18**(2), 77–95, (2002).

[20] Ya Xue, Xuejun Liao, Lawrence Carin, and Balaji Krishnapuram, 'Multi-task learning for classification with dirichlet process priors', *The Journal of Machine Learning Research*, **8**, 35–63, (2007).

[21] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai, 'A colorful approach to text processing by example', in *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pp. 495–504. ACM, (2013).