Learning Logic Programs by Discovering Higher-Order Abstractions

Céline Hocquette[1], Sebastijan Dumančić[2], Andrew Cropper[1]
[1]University of Oxford; [2]TU Delft
{celine.hocquette, andrew.cropper}@cs.ox.ac.uk; s.dumancic@tudelft.nl

UK Research and Innovation

UNIVERSITY OF OXFORD

TU Delft — Delft University of Technology

1 - Introduction

The goal of inductive logic programming (ILP) is to induce a program (a set of logical rules) that generalises training examples.

Using abstractions, such as *map*, *filter*, and *fold*, can allow us to learn smaller programs, which are often easier to learn than larger ones.

Example 1 (String transformation)
Positive example:

$$[l, o, g, i, c] \mapsto [L, O, G, I, C]$$

First-order program:

```
f(Input,Output) ←
        empty(Input), empty(Output)
f(Input,Output) ←
        head(Input,Head1),
        tail(Input,Tail1),
        uppercase(Head1,Head2),
        head(Output,Head2),
        tail(Output,Tail2),
        f(Tail1,Tail2)
```

Second-order program:

```
f(Input,Output) ←
        map(Input,Output,uppercase)
```

We introduce an approach that automatically discovers higher-order abstractions to improve learning performance.

Positive example:

$$[2, 6, 3, 8] \mapsto [3, 7, 4, 9]$$

First-order program:

```
g(Input,Output) ←
        empty(Input), empty(Output)
g(Input,Output) ←
        head(Input,Head1),
        tail(Input,Tail1),
        increment(Head1,Head2),
        head(Output,Head2),
        tail(Output,Tail2),
        f(Tail1,Tail2)
```

We introduce the abstraction *map*:

```
ho(Input,Output,Relation) ←
        empty(Input), empty(Output)
ho(Input,Output,Relation) ←
        head(Input,Head1),
        tail(Input,Tail1),
        Relation(Head1,Head2),
        head(Output,Head2),
        tail(Output,Tail2),
        ho(Tail1,Tail2,Relation)
```

We refactor the definitions *f* and *g* using *map*:

```
f(Input,Output) ←
        ho(Input,Output,uppercase)
g(Input,Output) ←
        ho(Input,Output,increment)
```

2 - Our approach (STEVIE)

Our approach works in two stages: *abstract* and *compress*.
In the *abstract* stage, STEVIE builds abstractions and instantiations.

Consider the rule:

f(A) ← head(A,B), one(B), tail(A,C), head(C,D), one(D)

Some abstractions of this rule are:

ho_1 *(A,X) ← X(A,B), one(B), tail(A,C), X(C,D), one(D)*
ho_2 *(A,X) ← head(A,B), X(B), tail(A,C), head(C,D), X(D)*
ho_3 *(A,X,Y) ← X(A,B), Y(B), tail(A,C), X(C,D), Y(D)*

Their instantiations are:

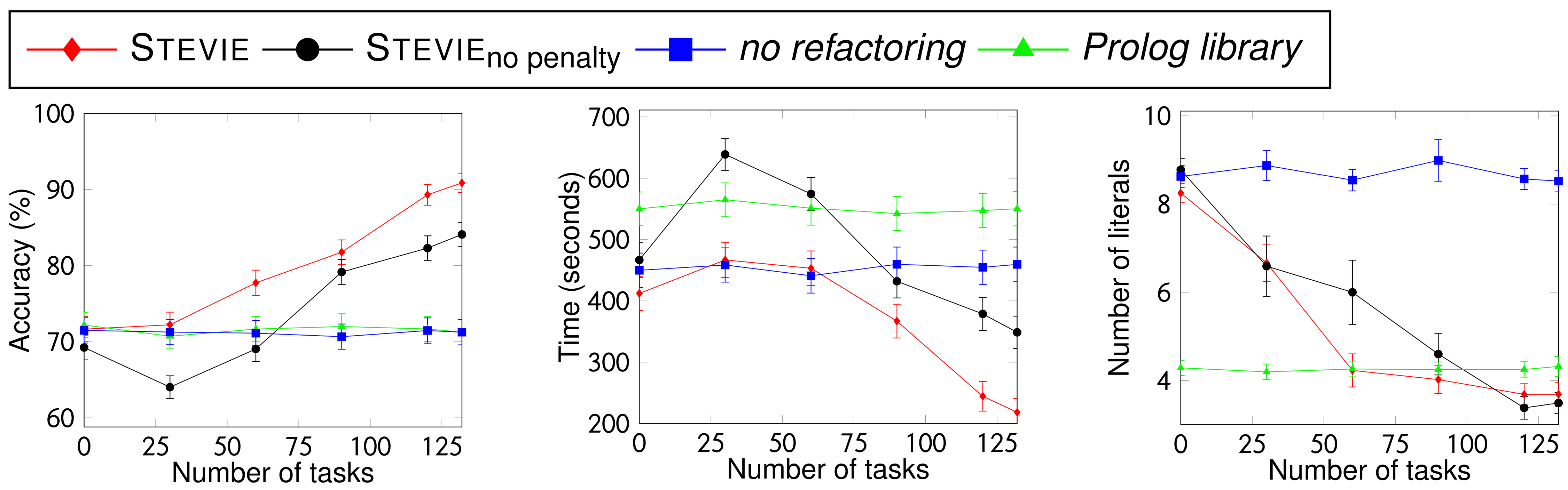
f(A) ← ho_1 *(A,head)*
f(A) ← ho_2 *(A,one)*
f(A) ← ho_3 *(A,head,one)*

In the *compress* stage, STEVIE searches for a subset of the abstractions which compresses the input program. STEVIE formulates this search problem as a constraint optimisation problem.

Theorem: STEVIE finds an optimal refactoring with respect to our objective function.

3 - Experiment

Q1 Can higher-order refactoring improve learning performance?



► **Higher-order refactoring can substantially improve learning performance.**

Task	Baseline	STEVIE
do5times	50 ± 0	**100** ± **0**
line1	50 ± 0	**100** ± **0**
line2	50 ± 0	**100** ± **0**
string1	50 ± 0	**100** ± **0**
string2	50 ± 0	**100** ± **0**
string3	50 ± 0	**100** ± **0**
string4	50 ± 0	**100** ± **0**
chessmapuntil	50 ± 0	**98** ± **1**
chessmapfilter	50 ± 0	**100** ± **0**
chessmapfilteruntil	50 ± 0	**98** ± **1**
droplastk	50 ± 0	**100** ± **0**
encryption	50 ± 0	**100** ± **0**
length	80 ± 12	**100** ± **0**
rotateN	50 ± 0	**100** ± **0**
waiter	50 ± 0	**100** ± **0**

Q2 Can higher-order refactoring improve performance across domains?

► **Learned abstractions transfer to different domains and higher-order refactoring can improve learning performance in different domains.**

4 - Conclusion and Limitation

► **An approach that discovers higher-order abstractions to refactor a logic program.**

Article