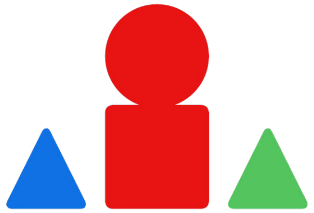
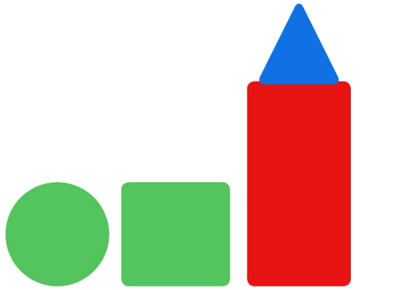
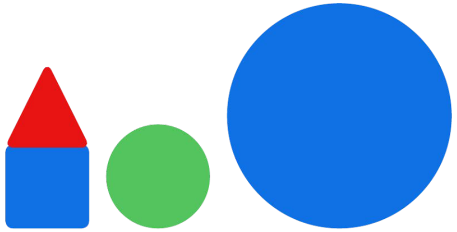
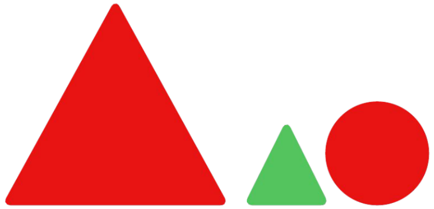
An Introduction to Inductive Logic Programming

Céline Hocquette

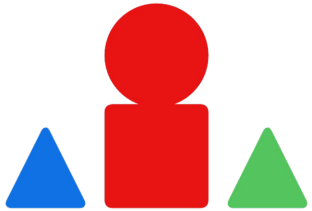
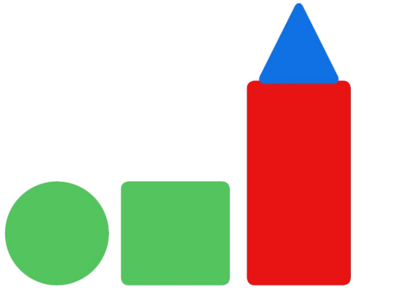
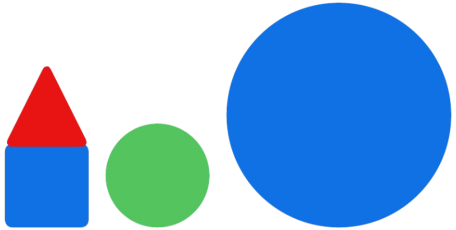
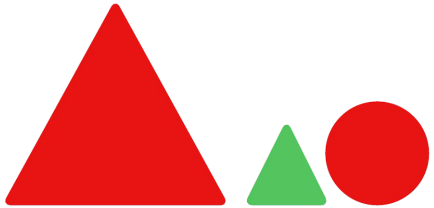
University of Oxford

slides available on my website: https://celinehocquette.github.io/

Game playing: Zendo

Positive structures	Negative structures
	
	

Game playing: Zendo

Positive structures	Negative structures
	
	

A zendo structure is positive if it contains a piece small and not blue in contact with another piece.

Encryption

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

Encryption

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr
learning	?

Encryption

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr
learning	?

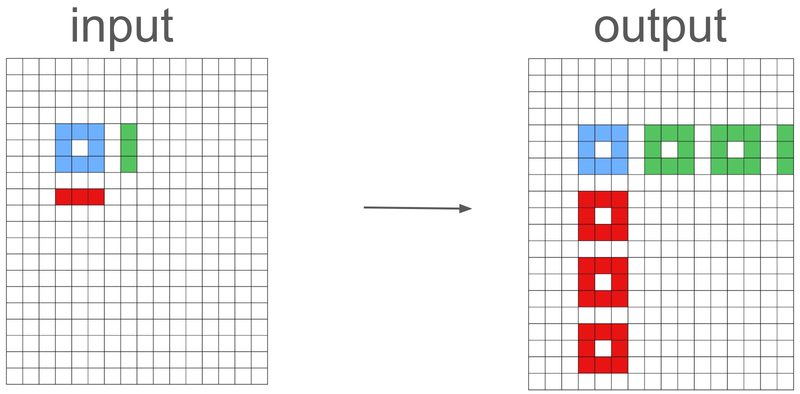
Add two to each element and reverse

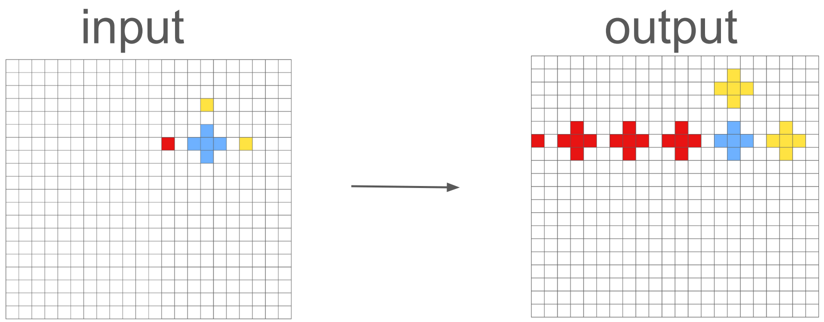
Encryption

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr
learning	ipkptcgn

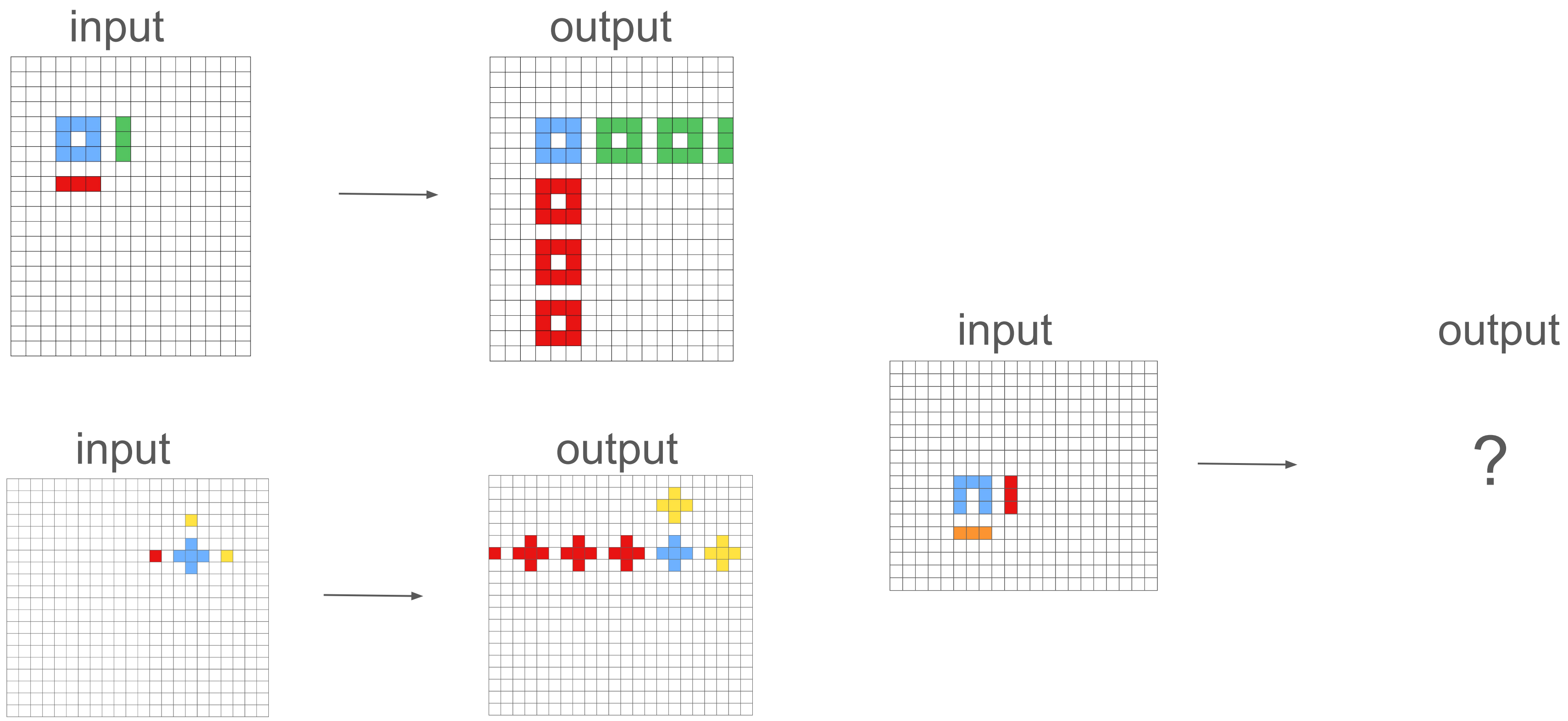
Add two to each element and reverse

Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]

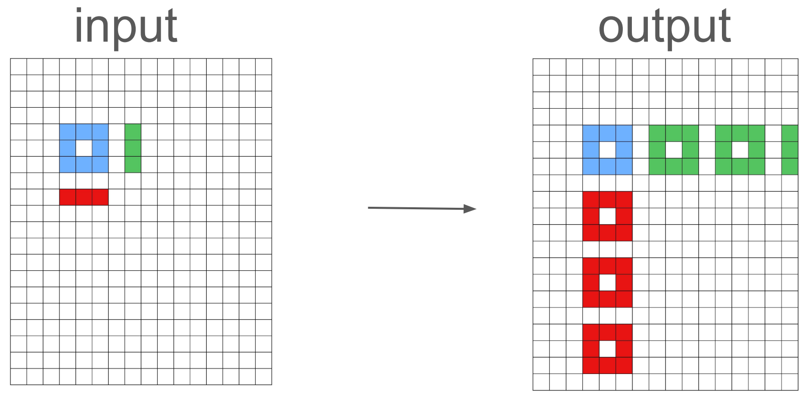


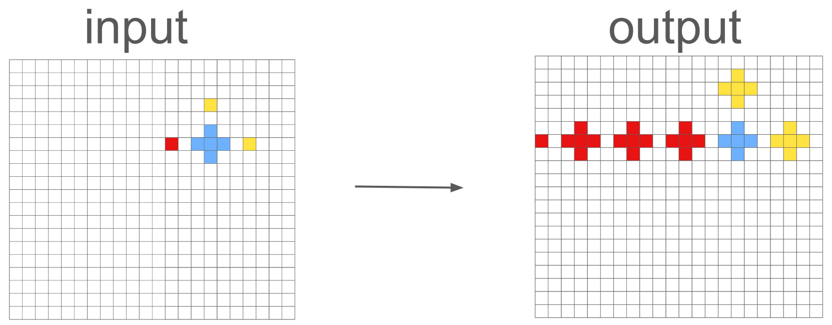


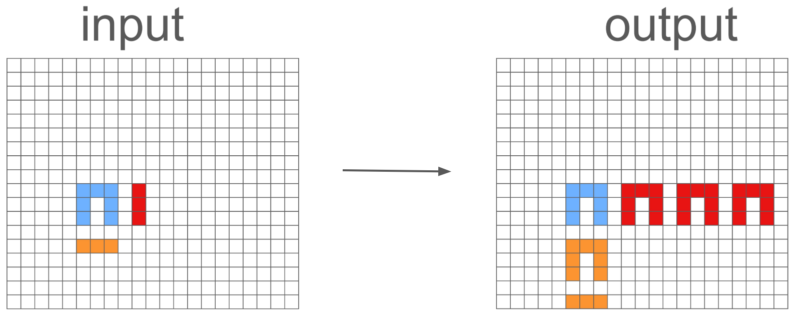
Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]



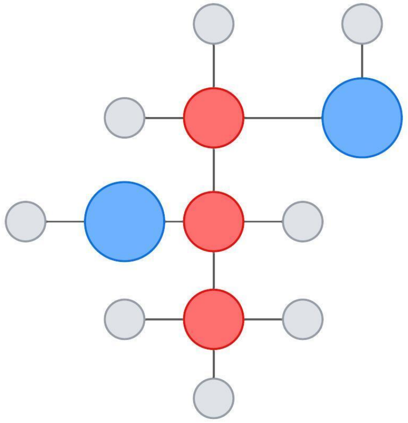
Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]



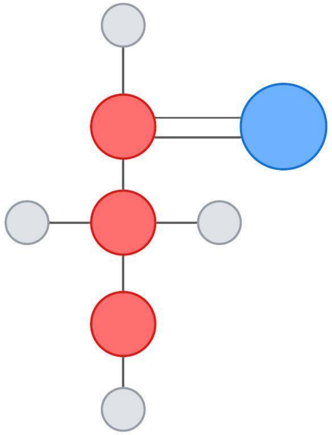




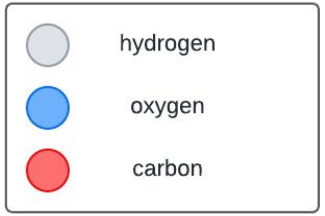
Drug design



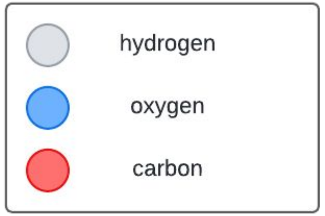
active molecule

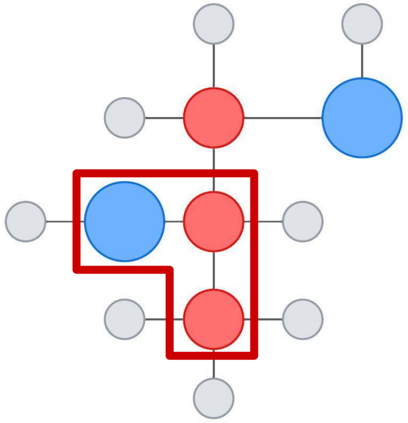


inactive molecule

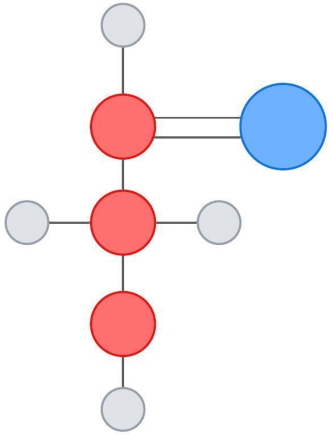


Drug design





active molecule



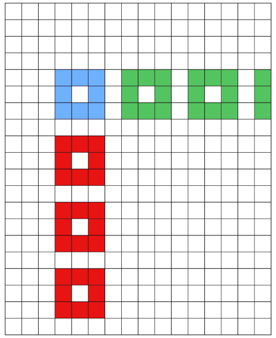
inactive molecule

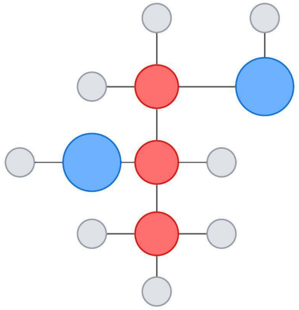
A molecule is active if it contains an oxygen atom bonded to a carbon atom, which is bonded to another carbon atom, by single bonds.

Let’s use machine learning to solve these problems!

What do we need?

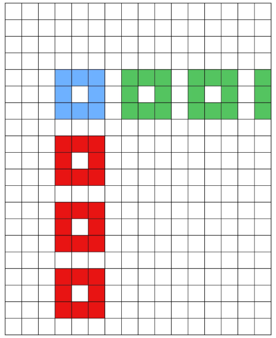
Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr





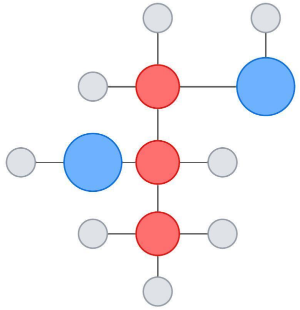
Introduction

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

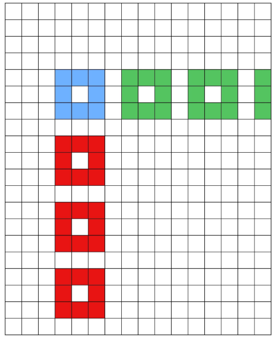


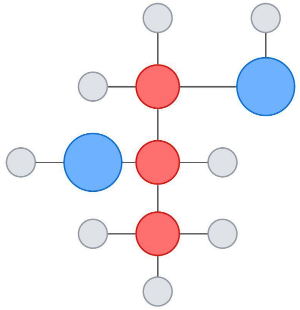
What do we need?

- learn from small amount of data



Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr



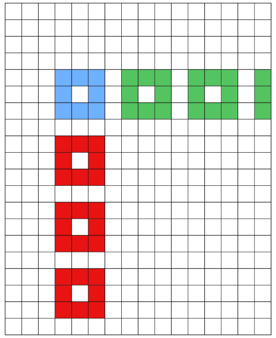


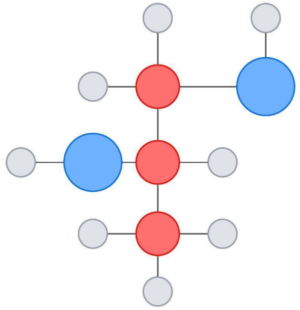
What do we need?

- learn from small amount of data
- learn interpretable programs

Introduction

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr



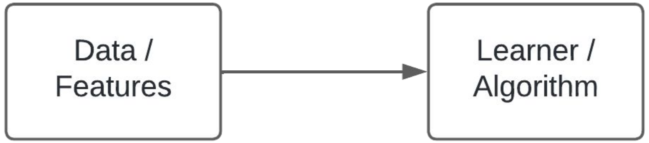


What do we need?

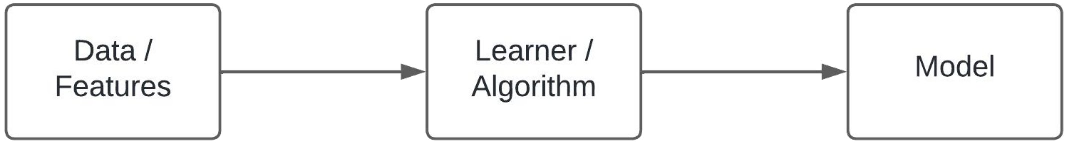
- learn from small amount of data
- learn interpretable programs
- learn from relational data

Machine Learning

Machine Learning



Machine Learning

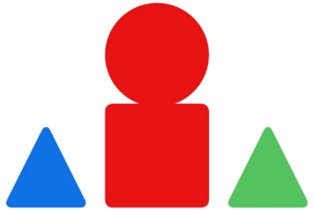


Machine Learning

Features:

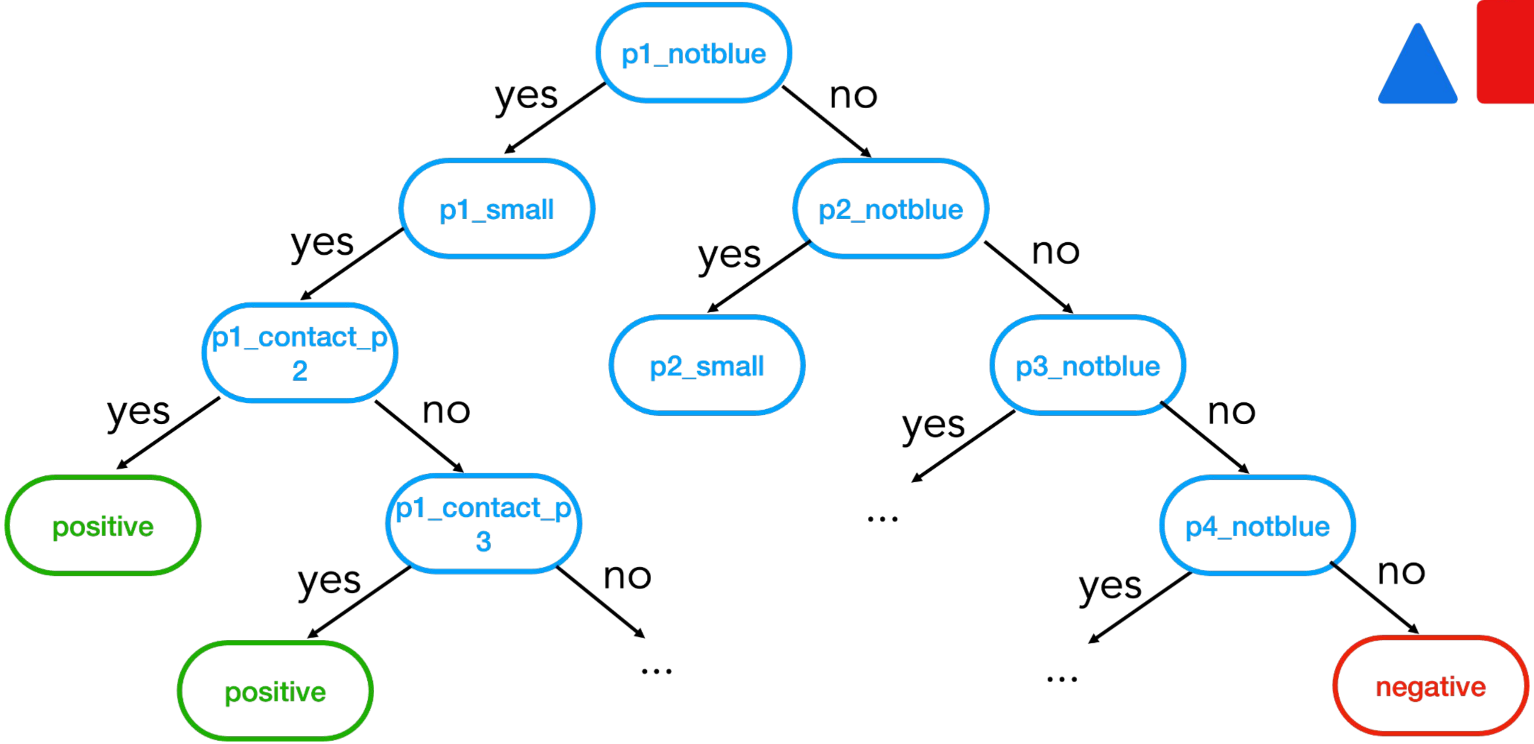
- blue green red notblue notgreen notred
  round square triangle rectangle
  small medium large
  contact_piece1 contact_piece2 contact_piece3 contact_piece4

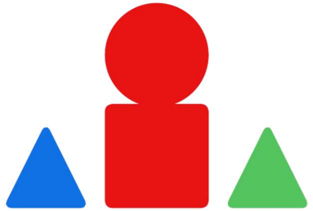
Machine Learning



	red	green	blue	triangle	rectangle	square	circle	contact_p1	contact_p2	contact_p3	contact_p4	small	medium	large
piece1	0	0	1	1	0	0	0	0	0	0	0	1	0	0
piece2	1	0	0	0	0	0	1	0	0	1	0	1	0	0
piece3	1	0	0	0	0	1	0	0	1	0	0	1	0	0
piece4	0	1	0	1	0	0	0	0	0	0	0	1	0	0

Zendo in decision tree





Machine Learning

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

Features:

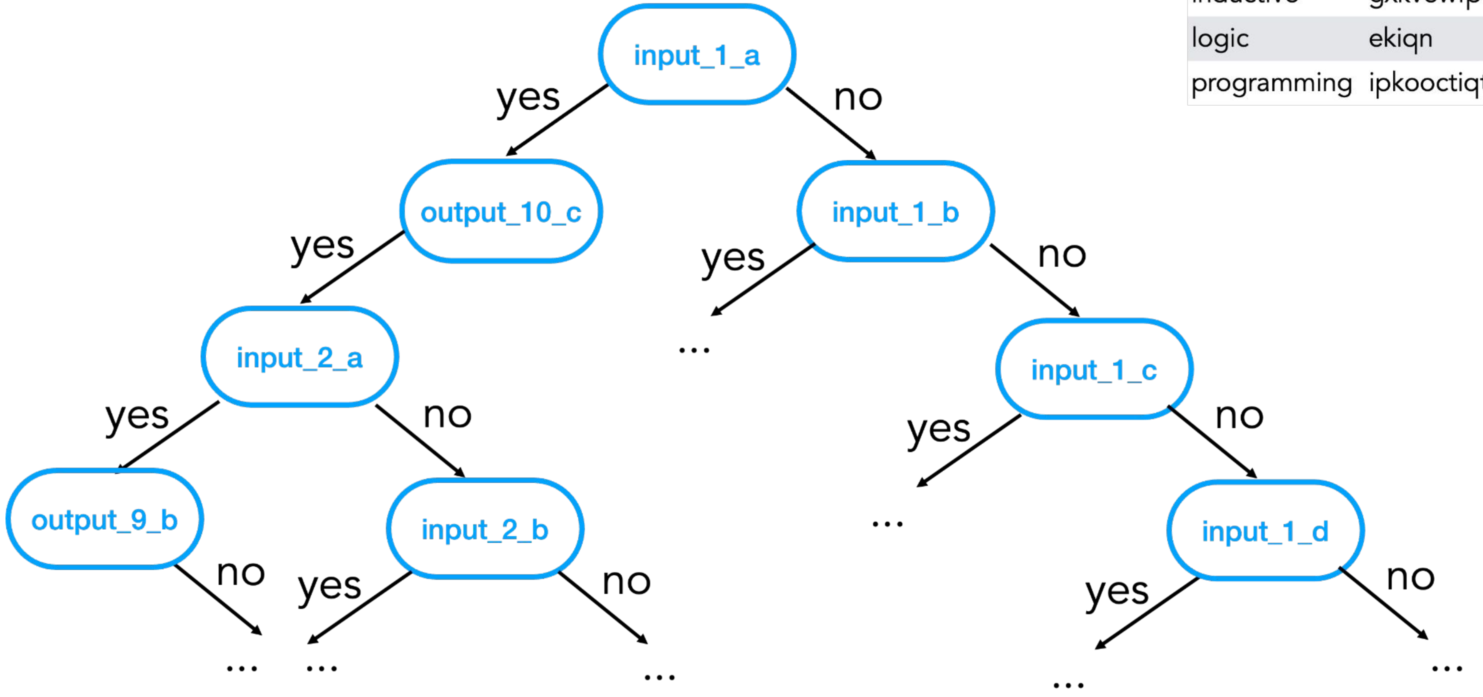
- input_1_a input_1_b input_1_c …
- input_2_a input_3_b input_2_c …
- input_3_a input_3_b input_3_c …

Machine Learning

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

	input_1_a	input_1_b	input_1_c	input_1_i	input_1_j	input_1_k	input_1_l	input_1_m	input_1_p
inductive	0	0	0	1	0	0	0	0	0
logic	0	0	0	0	0	0	1	0	0
programming	0	0	0	0	0	0	0	0	1

Encryption in decision tree



Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

Requirements:

- learn from small amount of data
- learn interpretable programs
- learn from relational data

These requirements are difficult for existing ML approaches.

In this presentation: an introduction to Inductive Logic Programming

Outline

1 - Introduction

2 - What is ILP?

3 - Representation language

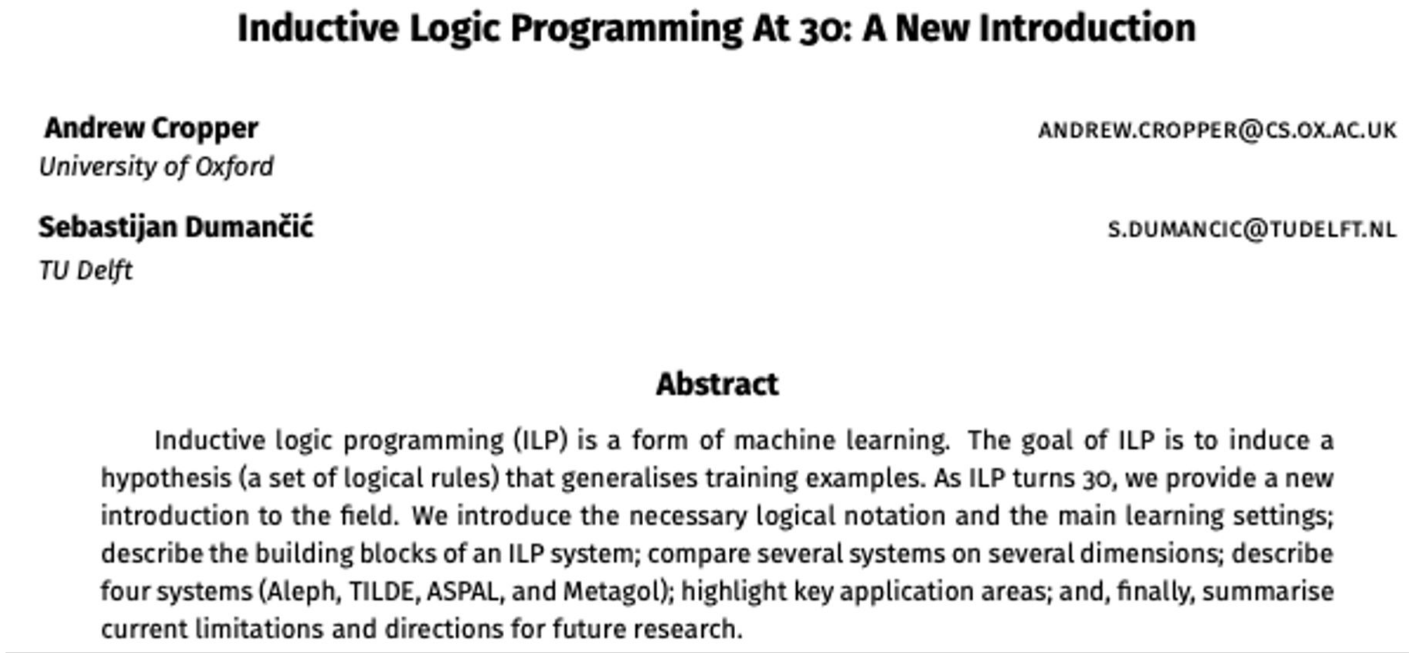
4 - Search techniques in ILP

5 - ILP features

6 - Case study: Popper

7 - Conclusion

More technical details



Inductive Logic Programming At 30: A New Introduction

Andrew Cropper ANDREW.CROPPER@CS.OX.AC.UK
University of Oxford

Sebastijan Dumančić S.DUMANCIC@TUDELFT.NL
TU Delft

Abstract

Inductive logic programming (ILP) is a form of machine learning. The goal of ILP is to induce a hypothesis (a set of logical rules) that generalises training examples. As ILP turns 30, we provide a new introduction to the field. We introduce the necessary logical notation and the main learning settings; describe the building blocks of an ILP system; compare several systems on several dimensions; describe four systems (Aleph, TILDE, ASPAL, and Metagol); highlight key application areas; and, finally, summarise current limitations and directions for future research.

What is Inductive Logic Programming?

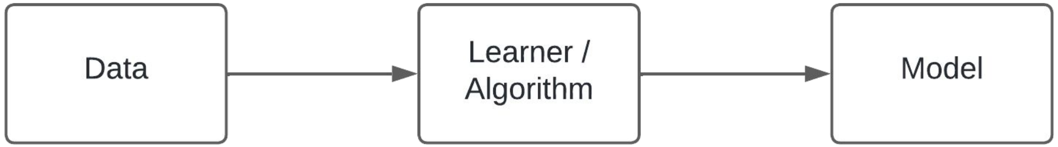
What is ILP?

ILP is a form of Machine Learning

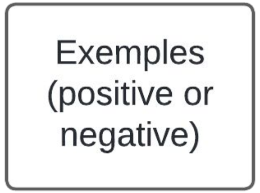
Inductive Logic Programming (ILP)

ILP = ML + logic

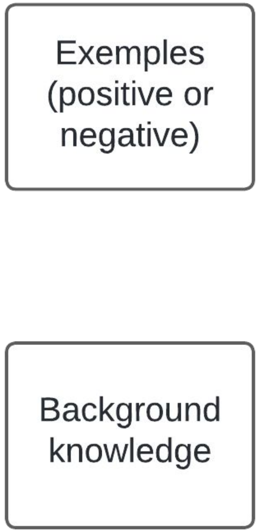
Machine Learning



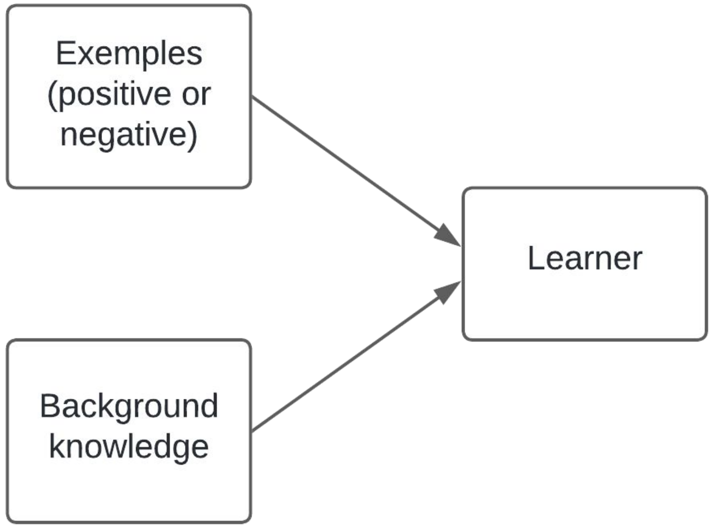
Inductive Logic Programming (ILP)



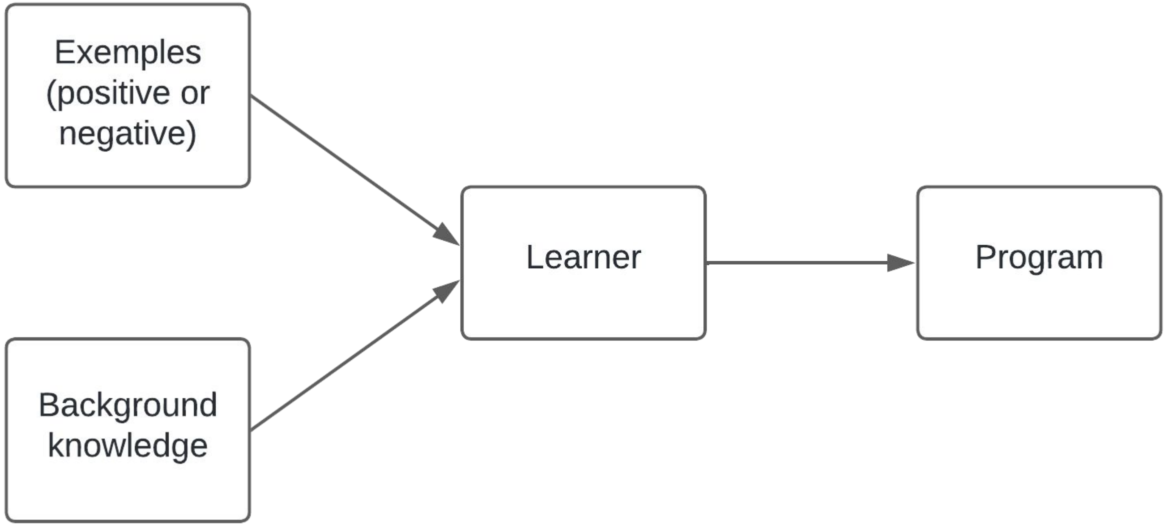
Inductive Logic Programming (ILP)



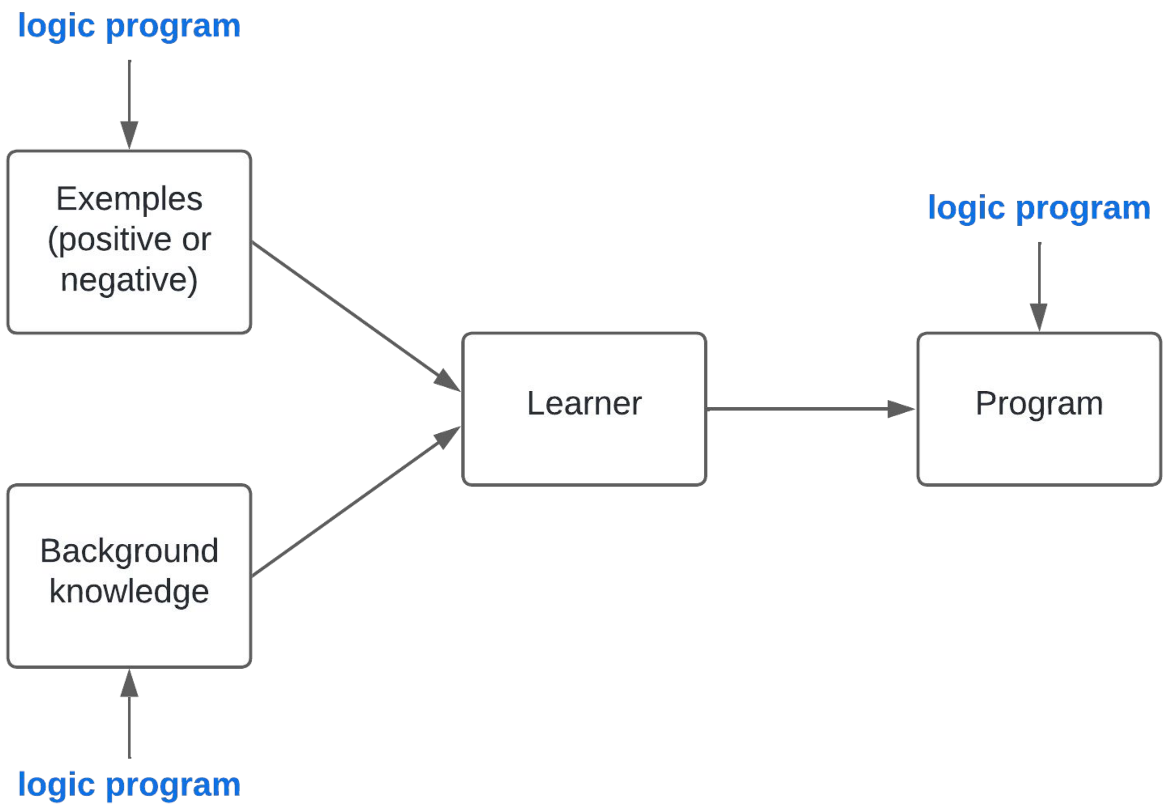
Inductive Logic Programming (ILP)



Inductive Logic Programming (ILP)

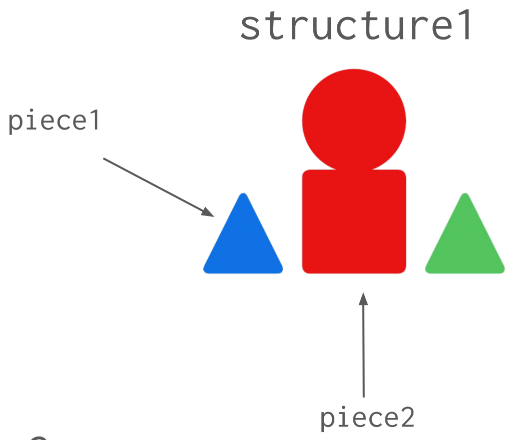


Inductive Logic Programming (ILP)



Logical refresher

Logic



constants:

`structure1, piece1, piece2, …`

Logic

variables:

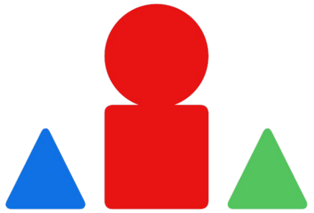
```
    Structure, Piece, A, B, C, …
```

Logic

predicates:

```
    blue/1, red/1, contact/2, distance/3, …
```

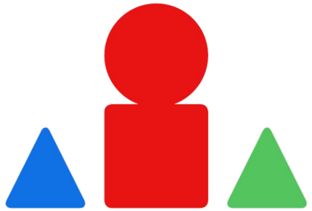
Logic



atoms:

```
    blue(piece1)
    red(piece2)
    triangle(Piece)
    contact(Piece, piece2)
    distance(A, B, 1)
    …
```

Logic



literal:

```
blue(piece1)
¬red(Piece)
triangle(piece2)
¬contact(Piece, piece2)
distance(A, B, 1)
…
```

Logic

a clause:

```
b1, …, bn → h1, …, hn.
```

Logic

a clause:

```
∀ Piece, blue(Piece), triangle(Piece) → good_piece(Piece).
```

Logic

a clause:

```
∀ Piece, blue(Piece), triangle(Piece) → good_piece(Piece).
```

if this side is true

then this side is true

Logic

a clause:

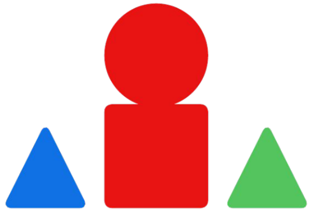
```
blue(Piece), triangle(Piece) → good_piece(Piece).
```

Logic

a clause:

```
good_piece(Piece) ← blue(Piece), triangle(Piece).
```

Logic



a program:

```
good_piece(Piece) ← blue(Piece), triangle(Piece).
good_piece(Piece) ← red(Piece), square(Piece).
```

Logic

```
blue(piece1).
good_piece(Piece) ← blue(Piece).
```

Logic

```
blue(piece1).
good_piece(Piece) ← blue(Piece).

good_piece(piece1).
```

Logic

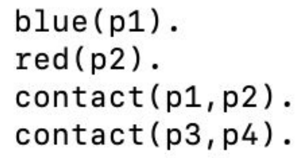
```
A: blue(piece1).
B: good_piece(Piece) ← blue(Piece).

C: good_piece(piece1).
```

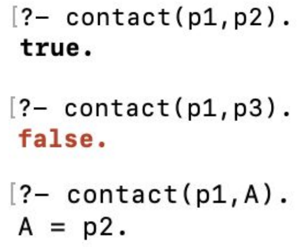
$\{A,B\} \models C$

Logic programming

programming paradigm based on logic



```
blue(p1).
red(p2).
contact(p1,p2).
contact(p3,p4).
```



```
[?- contact(p1,p2).
true.

[?- contact(p1,p3).
false.

[?- contact(p1,A).
A = p2.
```

Why logic programs?

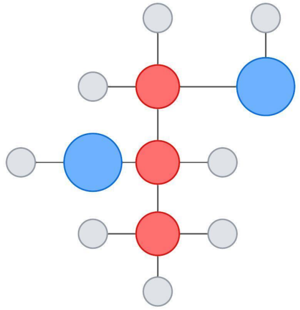
- relational

```
edge(bond_street, oxford_circus).

single_bond(atom1, atom2).

on_top(piece2, piece3).
aligned(piece1, piece3, piece4).
```

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Covent Garden
Queensway
Lancaster Gate
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple



Why logic programs?

- relational
- declarative

```
good_piece(Piece) ←
    blue(Piece),
    triangle(Piece),
    contact(Piece,Piece1),
    red(Piece1),
    square(Piece1).
```

can execute in any order
if any literal fails, the whole rule fails

Why logic programs?

- relational
- declarative

```
good_piece(Piece) ←
    blue(Piece),
    triangle(Piece),
    contact(Piece,Piece1),
    red(Piece1),
    square(Piece1).
good_piece(Piece) ←
    green(Piece),
    round(Piece).
```

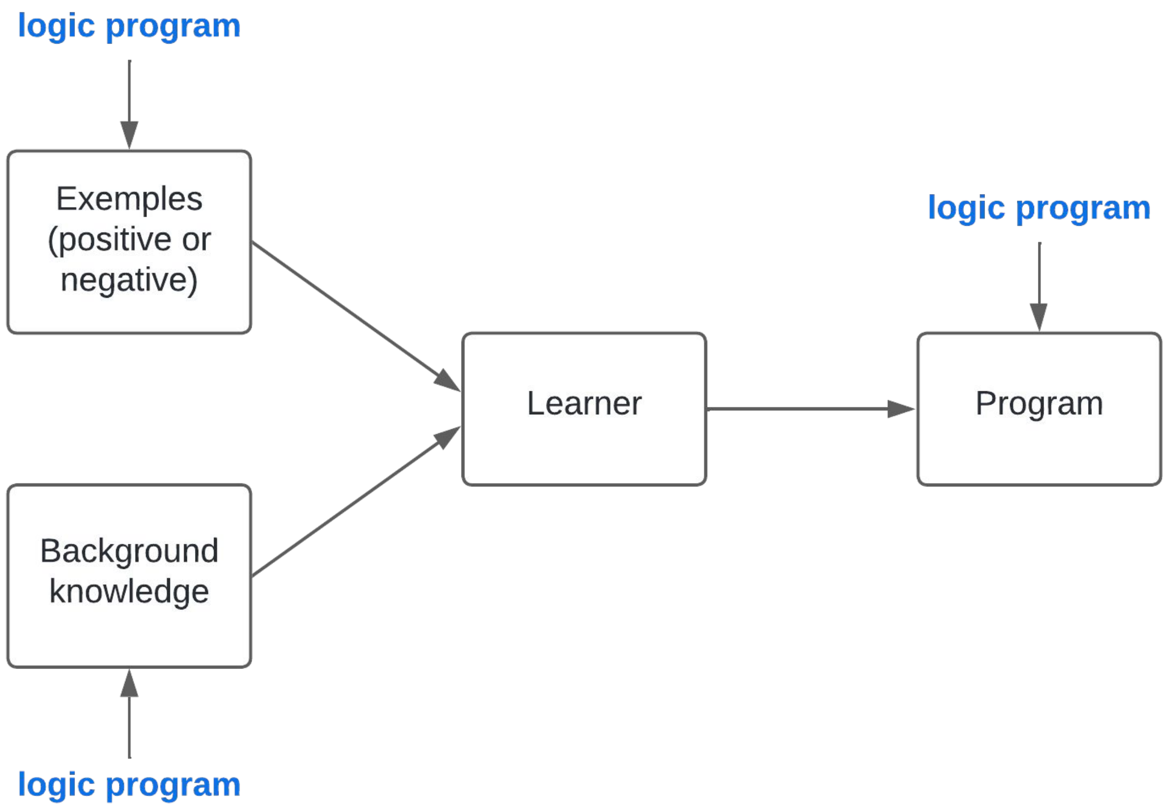
if any rule succeeds, the program succeeds

Why logic programs?

- relational
- declarative
- interpretable

```
good_piece(Piece) ←
    blue(Piece),
    triangle(Piece),
    contact(Piece,Piece1),
    red(Piece1),
    square(Piece1).
good_piece(Piece) ←
    green(Piece),
    round(Piece).
```

Inductive Logic Programming (ILP)



Inductive Logic Programming (ILP)
Learning from entailment

Given:

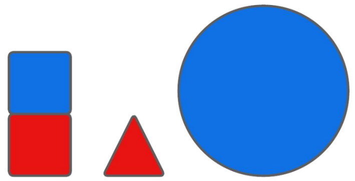
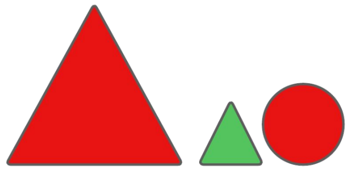
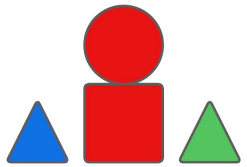
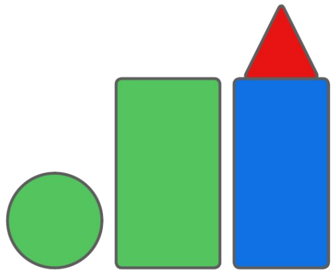
- positive examples E^+
- negative examples E^-
- background knowledge B

Find:

- H such that:
    - $\forall e \in E^+, B \cup H \models e$
    - $\forall e \in E^-, B \cup H \not\models e$

Let’s use ILP on our problems!

Zendo in ILP

Positive structures	Negative structures
	
	

Zendo in ILP

```
% positive examples
pos(zendo(structure1)).
pos(zendo(structure2)).
```

Zendo in ILP

```
% positive examples
pos(zendo(structure1)).
pos(zendo(structure2)).


% negative examples
neg(zendo(structure3)).
neg(zendo(structure4)).
```

Zendo in ILP

```
% positive examples
pos(zendo(structure1)).
pos(zendo(structure2)).



% negative examples
neg(zendo(structure3)).
neg(zendo(structure4)).
```

```
% background knowledge
piece(structure1, piece1).
piece(structure1, piece2).
piece(structure1, piece3).
piece(structure1, piece4).
blue(piece1).
red(piece2).
red(piece3).
blue(piece4).
square(piece1).
square(piece1).
triangle(piece1).
round(piece1).
small(piece2).
contact(p1,p2).
…
```

Zendo in ILP

```
% positive examples
pos(zendo(structure1)).
pos(zendo(structure2)).


% negative examples
neg(zendo(structure3)).
neg(zendo(structure4)).
```

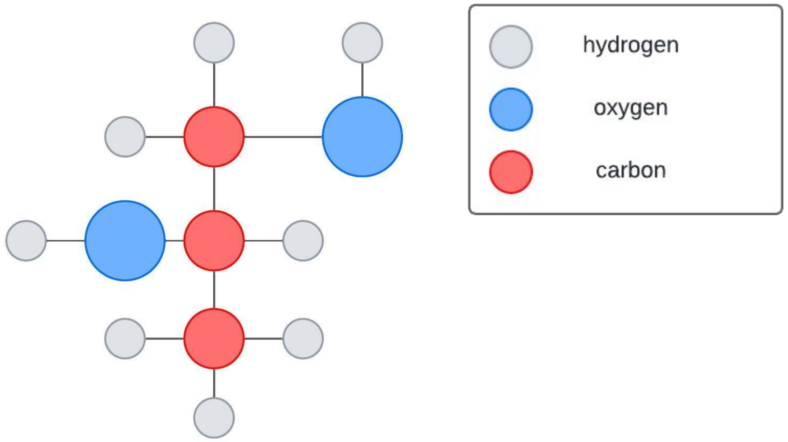
```
% background knowledge
piece(structure1, piece1).
piece(structure1, piece2).
piece(structure1, piece3).
piece(structure1, piece4).
blue(piece1).
red(piece2).
red(piece3).
blue(piece4).
square(piece1).
square(piece1).
triangle(piece1).
round(piece1).
small(piece2).
contact(p1,p2).
…
```

Learned program:

```
zendo(A) :-
      piece(A,C),
      contact(C,B),
      small(B),
      not_blue(B).
```

Drug design in ILP

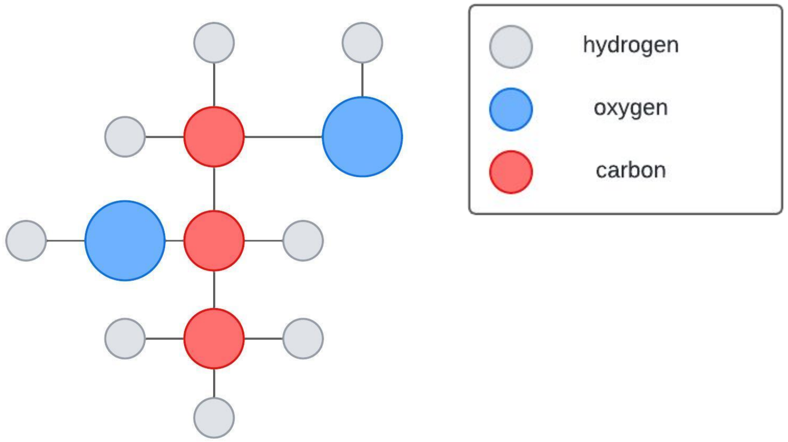
```
% positive examples
pos(active(molecule1)).
```



Drug design in ILP

```
% positive examples
pos(active(molecule1)).


% negative examples
neg(active(molecule2))
```

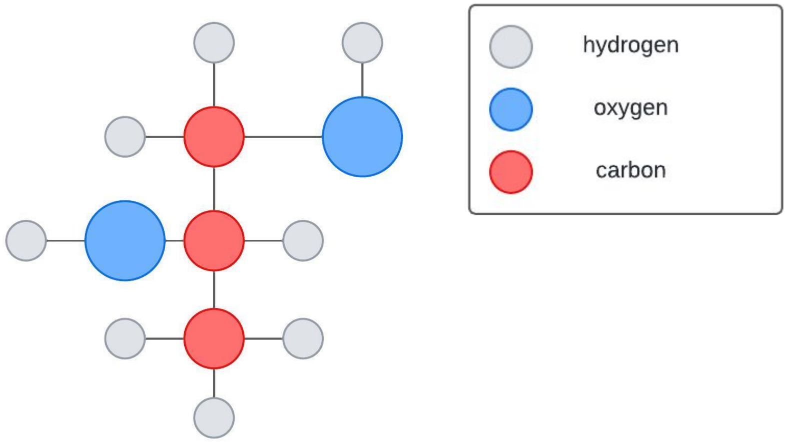


Drug design in ILP

```
% positive examples
pos(active(molecule1)).


% negative examples
neg(active(molecule2))
```

```
% background knowledge
atom(molecule1, atom1).
atom(molecule1, atom2).
atom(molecule1, atom3).
atom(molecule1, atom4).
hydrogen(atom1).
hydrogen(atom2).
oxygen(atom3).
carbon(atom4).
bond(atom1, atom3, single).
bond(atom3, atom4, single).
bond(A,B,C) :- bond(B,A,C).
…
```

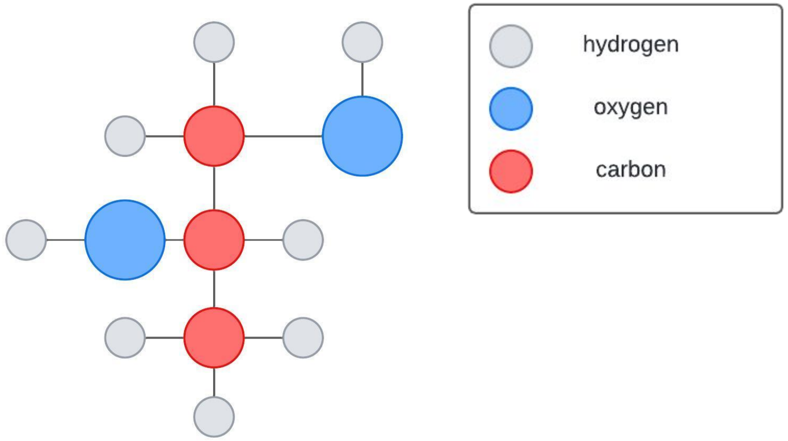


Drug design in ILP

```
% positive examples
pos(active(molecule1)).


% negative examples
neg(active(molecule2))
```

```
% background knowledge
atom(molecule1, atom1).
atom(molecule1, atom2).
atom(molecule1, atom3).
atom(molecule1, atom4).
hydrogen(atom1).
hydrogen(atom2).
oxygen(atom3).
carbon(atom4).
bond(atom1, atom3, single).
bond(atom3, atom4, single).
bond(A,B,C) :- bond(B,A,C).
…
```



```
% Learned program
active(Molecule) :-
        atom(Molecule,Atom1),
        oxygen(Atom1),
        atom(Molecule,Atom2),
        carbon(Atom2),
        atom(Molecule,Atom3),
        carbon(Atom3),
        bond(Atom1, Atom2, single),
        bond(Atom2, Atom3, single).
```

Inductive Logic Programming (ILP)

- generalise from a small amount of data

Inductive Logic Programming (ILP)

- generalise from a small amount of data
- learn interpretable programs

Inductive Logic Programming (ILP)

- generalise from a small amount of data
- learn interpretable programs
- learn from relational data

Questions?

Representation language

Which logic programming language?

- propositional logic

	red	green	blue	triangle	rectangle	square	circle	contact_p1	contact_p2	contact_p3	contact_p4	small	medium	large
piece1	0	0	1	1	0	0	0	0	0	0	0	1	0	0
piece2	1	0	0	0	0	0	1	0	0	1	0	1	0	0
piece3	1	0	0	0	0	1	0	0	1	0	0	1	0	0
piece4	0	1	0	1	0	0	0	0	0	0	0	1	0	0

Which logic programming language?

- propositional logic

```
good_structure :- piece1_small, piece1_notblue, contact_piece1_piece2.
```

Which logic programming language?

- propositional logic
    - limited expressivity (same as DT learners)
    - not relational
    - no recursion

Which logic programming language?

- first-order logic: intractable

∀A ∃B ∀C contact(A,B), green(B), left(A,C), blue(C) → small(A) □
¬right(A,B)

Which logic programming language?

- Horn logic: at most one positive literal
    - SLD-resolution
    - Turing complete

```
contact(A,B), green(B), left(A,C), blue(C) → good_piece(A)
```

Which logic programming language?

- Prolog

Advantages:

- Turing-complete
- list and complex data structure
- numerical reasoning

Disadvantage:

- not guaranteed to terminate

Which logic programming language?

- Datalog: definite programs without functional symbols and minor syntactic restrictions

Advantages:

- guaranteed to terminate
- sufficient for most problems

Disadvantage:

- not Turing complete (no function symbols)

Which logic programming language?

- monotonic vs non-monotonic logic

A logic is monotonic when adding knowledge to it does not reduce the logical consequences of that theory.

A logic is non-monotonic if some conclusions can be invalidated by adding more knowledge.

Which logic programming language?

```
blue(piece1).
good_piece(Piece) :- blue(Piece).
```

has consequences:

```
blue(piece1).
good_piece(piece1).
```

Which logic programming language?

```
blue(piece1).
good_piece(Piece) :- blue(Piece).
good_piece(Piece) :- red(Piece).
```

has consequences:

```
blue(piece1).
good_piece(piece1).
```

Which logic programming language?

Most non-monotonic programs use negation-as-failure (NAF) (Clark, 1977).

An atom is false if it cannot be proven true.

Which logic programming language?

```
blue(piece1).
good_piece(Piece) :- blue(Piece), not small(Piece).
```

has consequences:

```
blue(piece1).
good_piece(piece1).
```

Which logic programming language?

```
blue(piece1).
small(piece1).
good_piece(Piece) :- blue(Piece), not small(Piece).
```

has consequences:

```
blue(piece1).
```

Search techniques in ILP

How does ILP work?

The goal of ILP is to identify a program which correctly generalises the training examples among a search space.

What is the search space?

The search space is the set of all programs that may be output by the learner.

What is the search space?

The search space is defined by the *inductive bias*:

- syntactic bias
- semantic bias

Syntactic bias: Mode declarations

Specify which predicate symbols may appear in rules (and their types and directions)

Mode declarations

Specify which predicate symbols may appear in rules (and their types and directions)

```
modeh(*,zendo(+structure)).
modeb(*,piece(+structure,-piece)).
modeb(*,blue(+piece)).
modeb(*,contact(+piece,+piece)).
```

Mode declarations

Specify which predicate symbols may appear in rules (and their types and directions)

```
zendo(Structure) :-
    piece(Structure,Piece),
    blue(Piece)
```

✓

```
modeh(*,zendo(+structure)).
modeb(*,piece(+structure,-piece)).
modeb(*,blue(+piece)).
modeb(*,contact(+piece,+piece)).
```

Mode declarations

Specify which predicate symbols may appear in rules (and their types and directions)

```
modeh(*,zendo(+structure)).
modeb(*,piece(+structure,-piece)).
modeb(*,blue(+piece)).
modeb(*,contact(+piece,+piece)).
```

```
zendo(Structure) :-
     piece(Structure,Piece),
     blue(Piece)
```

✓

```
zendo(Structure) :-
     piece(Structure,Piece1),
     piece(Structure,Piece2),
     contact(Piece1,Piece2)
```

✓

Mode declarations

Specify which predicate symbols may appear in rules (and their types and directions)

```
zendo(Structure) :-
     piece(Structure,Piece),
     green(Piece)
```

❌

```
modeh(*,zendo(+structure)).
modeb(*,piece(+structure,-piece)).
modeb(*,blue(+piece)).
modeb(*,contact(+piece,+piece)).
```

Mode declarations

Specify which predicate symbols may appear in rules (and their types and directions)

```
modeh(*,zendo(+structure)).
modeb(*,piece(+structure,-piece)).
modeb(*,blue(+piece)).
modeb(*,contact(+piece,+piece)).
```

```
zendo(Structure) :-
    piece(Structure,Piece),
    green(Piece)
```

✗

```
zendo(Structure) :-
    piece(Structure,Piece1),
    contact(Structure,Piece1)
```

✗

Meta-rules

Specify the form of rules in programs

Meta-rules

Specify the form of rules in programs

```
P(A,B) :- Q(A,C), R(C,B)
```

Meta-rules

Specify the form of rules in programs

```
reachable(Node1,Node2) :-
    edge(Node1,Node3),
    edge(Node3,Node2).
```

✓

```
P(A,B) :- Q(A,C), R(C,B)
```

Meta-rules

Specify the form of rules in programs

```
P(A,B) :- Q(A,C), R(C,B)
```

```
reachable(Node1,Node2) :-
    edge(Node1,Node2),
    green(Node2).
```

✗

```
reachable(Node1,Node2) :-
    edge(Node1,Node3),
    edge(Node3,Node4),
    edge(Node4,Node2),
```

✗

What is the search space?

Choosing an appropriate inductive bias is essential!

too strong: we might exclude solutions, difficult to provide
too weak: large search space

How do we search the search space?

Generality ordering over the search space

Subsumption

```
C1 = zendo(U) ← piece(U,V), green(V)

C2 = zendo(A) ← piece(A,B), green(B), small(B)
```

Subsumption

```
C1 = zendo(U) ← piece(U,V), green(V)

C2 = zendo(A) ← piece(A,B), green(B), small(B)
```

C1 = {zendo(U), ¬piece(U,V),¬ green(V)}
C2 = {zendo(A), ¬piece(A,B), ¬green(B),¬small(B)}

Subsumption

```
C1 = zendo(U) ← piece(U,V), green(V)

C2 = zendo(A) ← piece(A,B), green(B), small(B)
```

$C1 = \{zendo(U), \neg piece(U,V), \neg green(V)\}$
$C2 = \{zendo(A), \neg piece(A,B), \neg green(B), \neg small(B)\}$

$\theta = \{A/U, B/V\}$
$\{zendo(U), \neg piece(U,V), \neg green(V)\}\theta \subseteq \{zendo(A), \neg piece(A,B), \neg green(B), \neg small(B)\}$

Subsumption

```
C1 = zendo(U) ← piece(U,V), green(V)

C2 = zendo(A) ← piece(A,B), green(B), small(B)
```

C1 = {zendo(U), ¬piece(U,V),¬ green(V)}
C2 = {zendo(A), ¬piece(A,B), ¬green(B),¬small(B)}

θ = {A/U, B/V}
{zendo(U), ¬piece(U,V),¬ green(V) } θ ⊆ {zendo(A), ¬piece(A,B), ¬green(B),¬small(B)}

C1 subsumes C2

Search techniques in ILP

C2 is more specific than C1 if C1 subsumes C2

```
C1 = zendo(Structure) :- piece(Structure,Piece), green(Piece)

C2 = zendo(Structure) :- piece(Structure,Piece), green(Piece), size(Piece,Size), small(Size)
```

C2 is more specific than C1: C2 entails fewer examples than C1

Search techniques in ILP

C2 is more general than C1 if C2 subsumes C1

```
C1 = zendo(Structure) :- piece(Structure,Piece), green(Piece)

C2 = {zendo(Structure) :- piece(Structure,Piece), green(Piece).
     {zendo(Structure) :- piece(Structure,Piece1), contact(Piece1,Piece2), blue(Piece2)
```

C2 is more general than C1: C2 entails more examples than C1

Subsumption lattice

$$\begin{tikzcd}
 & \texttt{good\_piece(A).} \arrow[dl] \arrow[d] \arrow[dr] & \\
\texttt{good\_piece(A) :- blue(A).} \arrow[d] & \texttt{good\_piece(A) :- small(A).} \arrow[dl] \arrow[d] & \texttt{good\_piece(A) :- round(A).} \arrow[d] \\
\texttt{good\_piece(A) :- blue(A),small(A).} \arrow[d] & \dots \arrow[dd] & \texttt{good\_piece(A) :- round(A),contact(A,B).} \arrow[d] \\
\dots \arrow[dr] & & \dots \arrow[dl] \\
 & \begin{array}{c}\texttt{good\_piece(A) :-}\\ \texttt{blue(A),small(A),round(A),}\\ \texttt{contact(A,B),large(B),rectangle(B).}\end{array} &
\end{tikzcd}$$

Top-down

Start with a general hypothesis and iteratively specialise it

FOIL, Tilde

Top-down

Start with a general hypothesis and iteratively specialise it

1. find a rule which covers some positive examples, by using heuristics to guide the search.

Top-down

Start with a general hypothesis and iteratively specialise it

1. find a rule which covers some positive examples, by using heuristics to guide the search.

```
good_piece(A) :- blue(A)
```

Top-down

Start with a general hypothesis and iteratively specialise it

1. find a rule which covers some positive examples, by using heuristics to guide the search.

```
good_piece(A) :- blue(A), small(A)
```

Top-down

Start with a general hypothesis and iteratively specialise it

1. find a rule which covers some positive examples, by using heuristics to guide the search.

```
good_piece(A) :- blue(A), small(A), number_contact(A,X)
```

Top-down

Start with a general hypothesis and iteratively specialise it

1. find a rule which covers some positive examples, by using heuristics to guide the search.

```
good_piece(A) :- blue(A), small(A), number_contact(A,X), X>3.
```

$\swarrow$

no information gain but needed!

Top-down

Start with a general hypothesis and iteratively specialise it

1. find a rule which covers some positive examples, by using heuristics to guide the search.
2. repeat step 1 on the uncovered positive examples

Top-down

Start with a general hypothesis and iteratively specialise it

Advantages:
- recursion

Disadvantages:
- inefficient
- not optimal

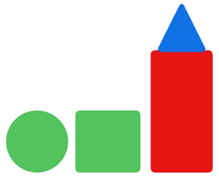
Bottom-up

Start with a specific hypothesis and iteratively generalise it

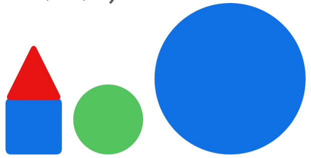
CIGOL, GOLEM

Bottom-up

Start with a specific hypothesis and iteratively generalise it



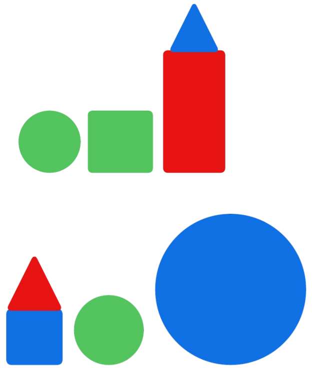
```
good_piece(A) :-
piece(A,B),green(B),small(B),round(B),piece(A,C),square(C),green(C),
piece(A,D),rectangle(D),large(D),red(D),piece(A,E),triangle(E),...
```



```
good_piece(A) :-
piece(A,B),square(B),small(B),blue(B),piece(A,C),triangle(C),red(C),
piece(A,D),round(D),large(D),blue(D),piece(A,E),round(E),...
```

Bottom-up

Start with a specific hypothesis and iteratively generalise it



```
good_piece(A) :-
piece(A,B),green(B),small(B),round(B),piece(A,C),square(C),piece(A,D)
,blue(A,D)
```

Bottom-up

Start with a specific hypothesis and iteratively generalise it

Advantages:
- fast

Disadvantages:
- optimality
- recursion
- predicate invention

Bidirectional search

bottom-up + top-down

Bidirectional search

bottom-up + top-down

1. Bottom-up: find the most specific rule R for each positive example
2. Top-down: search the generalisations of R

Progol, Aleph

Bidirectional search

bottom-up + top-down

1. Bottom-up: find the most specific rule R for each positive example
2. Top-down: search the generalisations of R

Advantages:
- fast
- large programs

Bidirectional search

bottom-up + top-down

1. Bottom-up: find the most specific rule R for each positive example
2. Top-down: search the generalisations of R

Advantages:
- fast
- large programs

Disadvantages:
- overfitting
- recursion
- predicate invention

Meta-level

Search all over

Metagol, ASPAL, ILASP, HexMIL, δILP, Popper,

Meta-level

Search all over

Delegate the search to a solver (SAT / ASP / SMT)

Meta-level

Search all over

Delegate the search to a solver (SAT / ASP / SMT)

Advantages:

- recursion
- optimality
- completeness

Meta-level

Search all over

Delegate the search to a solver (SAT / ASP / SMT)

Advantages:

- recursion
- optimality
- completeness

Disadvantages:

- small domains

Questions?

ILP Features

Recursion

connected(A,B) ⟵ edge(A,B).

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Lancaster Gate
Queensway
Covent Garden
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple

Recursion

connected(A,B) ⟵ edge(A,B).

connected(A,B) ⟵ edge(A,C),edge(C,B).

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Lancaster Gate
Queensway
Covent Garden
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple

Recursion

connected(A,B) ⟵ edge(A,B).

connected(A,B) ⟵ edge(A,C),edge(C,B).

connected(A,B) ⟵ edge(A,C),edge(C,D),edge(D,B).

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Lancaster Gate
Queensway
Covent Garden
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple

Recursion

connected(A,B) ⟵ edge(A,B).

connected(A,B) ⟵ edge(A,C),edge(C,B).

connected(A,B) ⟵ edge(A,C),edge(C,D),edge(D,B).

connected(A,B) ⟵ edge(A,C),edge(C,D),edge(D,E),edge(E,B).

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Lancaster Gate
Queensway
Covent Garden
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple

Recursion

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Queensway
Lancaster Gate
Covent Garden
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple

connected(A,B) ⟵ edge(A,B).

connected(A,B) ⟵ edge(A,C),edge(C,B).

connected(A,B) ⟵ edge(A,C),edge(C,D),edge(D,B).

connected(A,B) ⟵ edge(A,C),edge(C,D),edge(D,E),edge(E,B).

- Cannot generalise to arbitrary depth
- Difficult to learn because of its size

Recursion

Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Tottenham Court Road
Holborn
Notting Hill Gate
Lancaster Gate
Queensway
Covent Garden
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple

connected(A,B) ⟵ edge(A,B).

connected(A,B) ⟵ edge(A,C),connected(C,B).

- Generalises to any size
- Smaller and therefore easier to learn (needs fewer examples)

Predicate Invention

Automatically invent new symbols

Predicate Invention

Automatically invent new symbols
1 - write shorter programs
2 - express new concepts

Predicate Invention: write shorter programs

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
```

Predicate Invention: write shorter programs

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
```

Predicate Invention: write shorter programs

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),father(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- father(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),mother(C,D),mother(D,B).
```

- Difficult to learn because of its size
- Needs many examples

Predicate Invention: write shorter programs

```
greatgrandparent(A,B):- inv(A,C),inv(C,D),inv(D,B).
inv(A,B):- mother(A,B).
inv(A,B):- father(A,B).
```

Predicate Invention: write shorter programs

```
greatgrandparent(A,B):- inv(A,C),inv(C,D),inv(D,B).
inv(A,B):- mother(A,B).
inv(A,B):- father(A,B).
```

parent relation

Predicate Invention: write shorter programs

```
greatgrandparent(A,B):- inv(A,C),inv(C,D),inv(D,B).
inv(A,B):- mother(A,B).
inv(A,B):- father(A,B).
```

- Shorter and therefore easier to learn
- Needs fewer examples

Predicate Invention: express new concepts

Find the maximum value of a list and add it to every element

Predicate Invention: express new concepts

Find the maximum value of a list and add it to every element

```
f(A,B):- inv1(A,Max), …

inv1(A,B):- head(A,B), empty(B).
inv1(A,B):- head(A,B), inv1(A,C), B>C.
inv1(A,B):- head(A,C), inv1(A,B), B=<D.
```

Predicate Invention: express new concepts

Find the maximum value of a list and add it to every element

```
f(A,B):- inv1(A,Max), inv2(A,Max,B).

inv1(A,B):- head(A,B), empty(B).
inv1(A,B):- head(A,B), inv1(A,C), B>C.
inv1(A,B):- head(A,C), inv1(A,B), B=<D.

inv2(A,Max,B):- empty(A), empty(B).
inv2(A,Max,B):- head(A,H1), add(H1,Max,H2), tail(A,T1), head(B,H2), inv2(T1,Max,T2), tail(B,T2).
```

Higher-order programs

higher-order relation: a relation which takes another relation as argument
eg: `fold`, `map`, `filter`, `count`

Higher-order programs

Input	Output
logic	LOGIC
program	PROGRAM
learning	LEARNING

Higher-order programs

Input	Output
logic	LOGIC
program	PROGRAM
learning	LEARNING

First-order program:

```
map_uppercase(A,B) ← empty(A),empty(B).
map_uppercase(A,B) ← head(A,C),uppercase(C,D),tail(A,E),map_uppercase(E,F),head(B,D),tail(B,F).
```

Higher-order programs

Input	Output
logic	LOGIC
program	PROGRAM
learning	LEARNING

Second-order program:

```
map_uppercase(A,B) ← map(A,B,uppercase).
```

Higher-order programs

Input	Output
logic	LOGIC
program	PROGRAM
learning	LEARNING

Second-order program:

```
map_uppercase(A,B) ← map(A,B,uppercase).
```

- Shorter and therefore easier to learn
- Needs fewer examples to learn it

Higher-order programs + invention

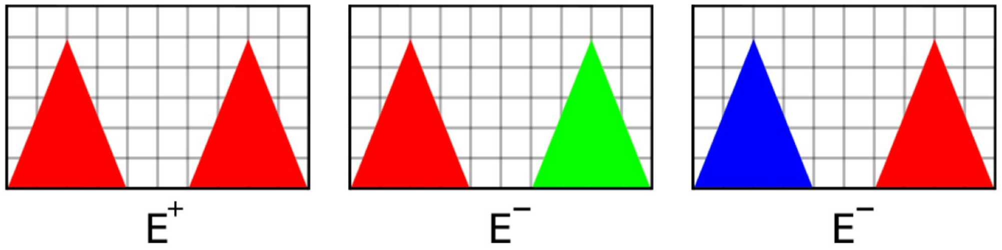
Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

Higher-order programs + invention

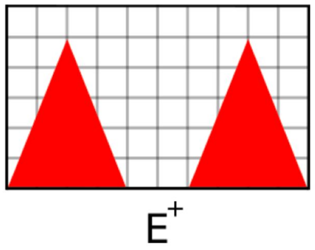
Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

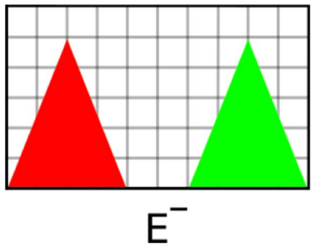
```
str_transformation(Input,Output)←
     map(inv_1,Input,String),
     reverse(String,Output).
inv_1(InputChar, OuputChar)←
     ord(InputChar,Number1),
     succ(Number1,Number2),
     succ(Number1,Number2),
     chr(Number2,OutputChar).
```

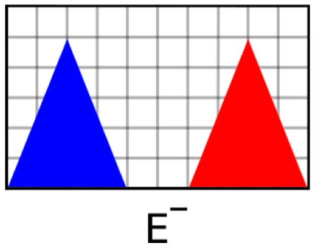
Negation



Negation

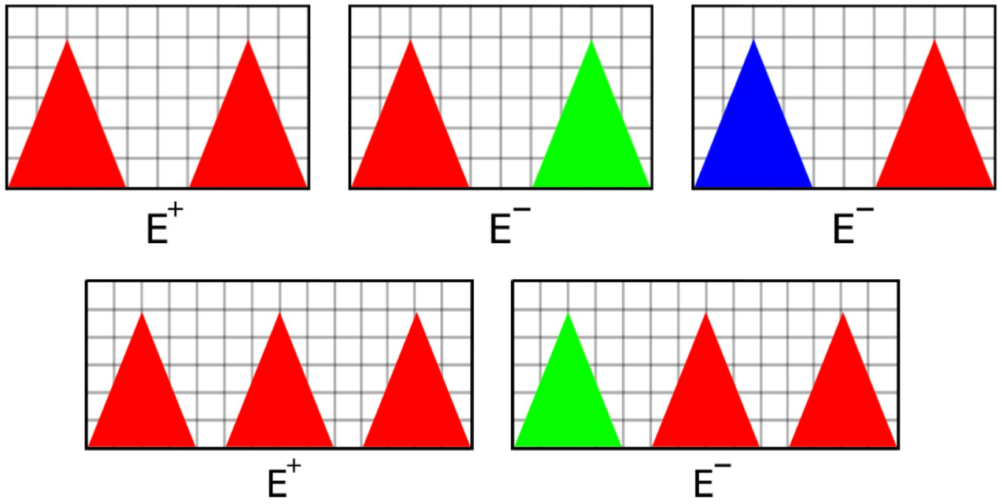




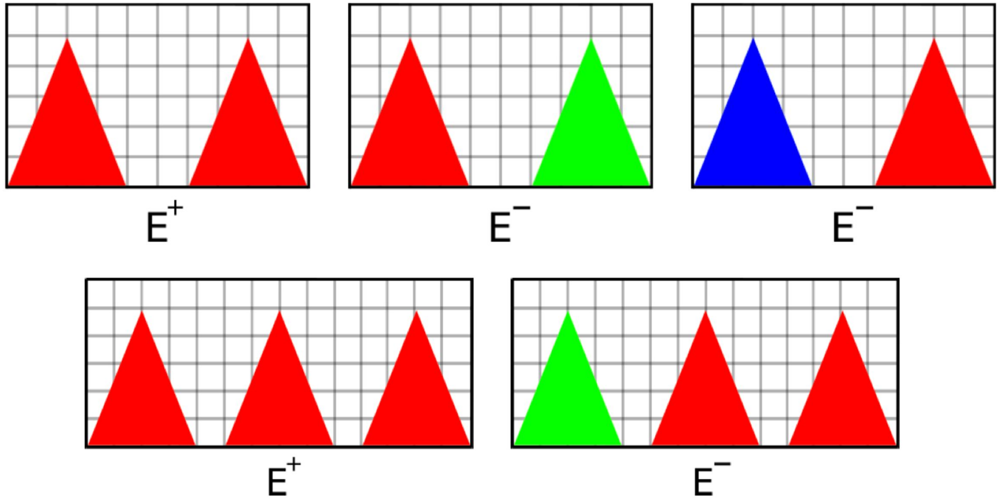


```
zendo(A) :- cone(A,C1), red(C1), cone(C2), red(C2), all_diff(C1,C2).
```

Negation

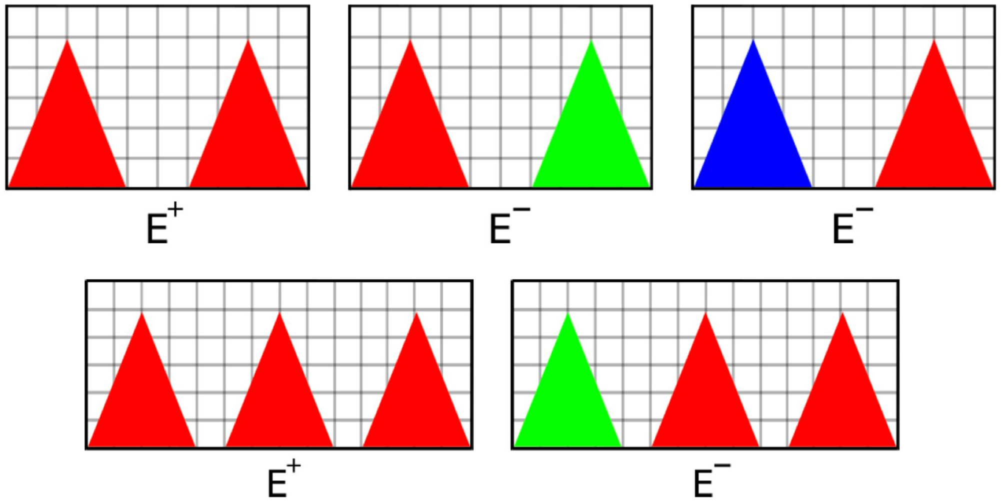


Negation



```
zendo(A) :- cone(A,C1), red(C1), cone(C2), red(C2), all_diff(C1,C2).
zendo(A) :- cone(A,C1), red(C1), cone(C2), red(C2), cone(C3), red(C3), all_diff(C1,C2,C3).
```

Negation



```
zendo(A) :- not inv_1(A).
inv_1(A) :- cone(A), not red(A).
```

all the cones are red

Learning optimal programs: textually minimal programs

```
zendo(A)← count(A,blue,2).
zendo(A)← count(A,blue,4).
zendo(A)← count(A,blue,6).
zendo(A)← count(A,blue,8).
…
```

Learning optimal programs: textually minimal programs

```
zendo(A)← count(A,blue,B), even(B).
```

Learning optimal programs: textually minimal programs

```
zendo(A)← count(A,blue,B), even(B).
```

- easier to interpret
- not necessarily better generalisation over unseen data!

Learning optimal programs: efficient programs

Input	Output
sheep	e
alpaca	a
chicken	?

Learning optimal programs: efficient programs

Input	Output
sheep	e
alpaca	a
chicken	c

Learning optimal programs: efficient programs

Input	Output
sheep	e
alpaca	a
chicken	c

```
f(A,B):- head(A,B),tail(A,C),element(C,B).
f(A,B):- tail(A,C),f(C,B).
```

Learning optimal programs: efficient programs

Input	Output
sheep	e
alpaca	a
chicken	c

```
f(A,B):- head(A,B),tail(A,C),element(C,B).
f(A,B):- tail(A,C),f(C,B).
```

$\searrow$

$O(n^2)$

Learning optimal programs: efficient programs

Input	Output
sheep	e
alpaca	a
chicken	c

```
f(A,B):- mergesort(A,C),inv1(C,B).
inv1(A,B):- head(A,B),tail(A,C),head(C,B).
inv1(A,B):- tail(A,C),inv1(C,B).
```

Learning optimal programs: efficient programs

Input	Output
sheep	e
alpaca	a
chicken	c

```
f(A,B):- mergesort(A,C),inv1(C,B).
inv1(A,B):- head(A,B),tail(A,C),head(C,B).
inv1(A,B):- tail(A,C),inv1(C,B).
```

O(n logn)

Noisy data

- Noisy examples
- Noisy BK

Noisy data: noisy examples

Most ILP systems support noisy examples

Noisy data: noisy examples

Most ILP systems support noisy examples:

- sequential covering approaches

Noisy data: noisy examples

Most ILP systems support noisy examples:

- sequential covering approaches
- divide-and-conquer approaches

Noisy data: noisy examples

Most ILP systems support noisy examples:

- sequential covering approaches
- divide-and-conquer approaches
- meta-level approaches

Noisy data: noisy examples with meta-level approaches

Relax the ILP solution definition
Leverage solver optimisations approaches to find a program with the best coverage

Noisy data: noisy examples with meta-level approaches

Relax the ILP solution definition
Leverage solver optimisations approaches to find a program with the best coverage

Which cost function?

Noisy data: noisy examples

Minimal description length: trade-off model complexity and the fit with the data

Noisy data: noisy examples

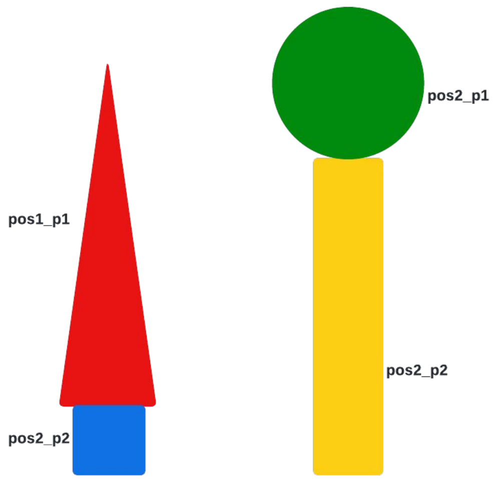
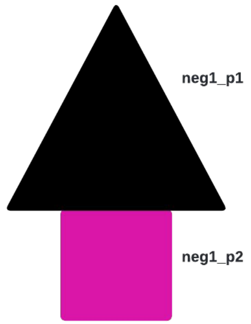
Minimal description length: trade-off model complexity and the fit with the data

$$\mathrm{mdl}(p) = \mathrm{fp}(p) + \mathrm{fn}(p) + \mathrm{size}(p)$$

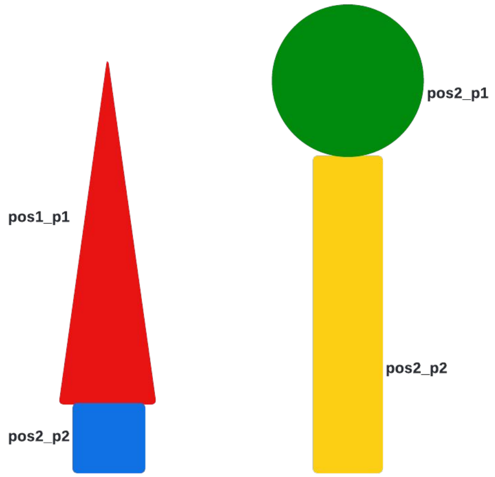
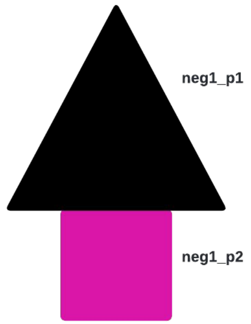
Noisy data: noisy BK

Difficult for current ILP systems!

Learning programs with numerical values

Positive examples	Negative examples
	

Learning programs with numerical values

Positive examples	Negative examples
	

zendo(A) ← piece(A,B),contact(B,C),size(C,D),geq(D,**7**).

Learning programs with numerical values

Challenges:

- infinite domains

Learning programs with numerical values

Challenges:

- infinite domains

```
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c1(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c2(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c3(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c4(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c5(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c6(E).
…
```

Learning programs with numerical values

Challenges:

- infinite domains

```
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c1(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c2(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c3(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c4(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c5(E).
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,E),c6(E).
…
```

```
zendo(A)← piece(A,B),contact(B,C),size(C,D),geq(D,Var),constant(Var)
```

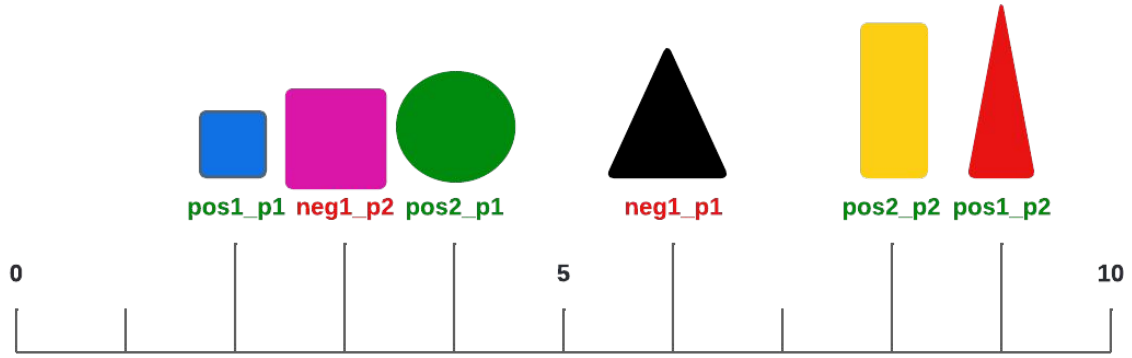
Learning programs with magic values, Céline Hocquette and Andrew Cropper, Machine learning, 2022.

Learning programs with numerical values

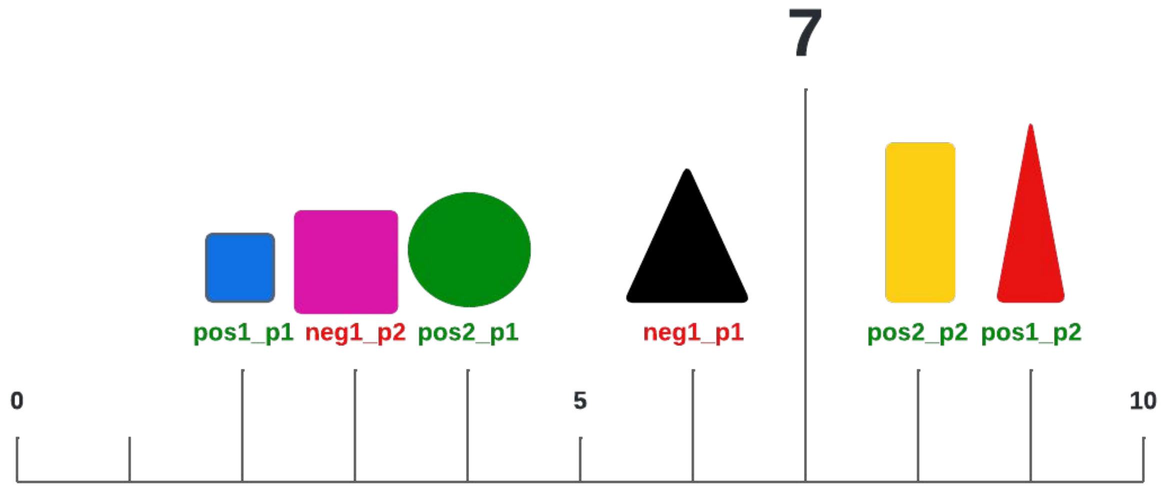
Challenges:

- infinite domains
- numerical reasoning considering all of the examples

Learning programs with numerical values



Learning programs with numerical values



Learning programs with numerical values

```
pharma(A):- zinc(A,B), hacc(A,C), dist(A,B,C,D), leq(D,4.18), geq(D,2.22).
pharma(A):- hacc(A,C), hacc(A,E), dist(A,B,C,D), geq(D,1.23), leq(D,3.41).
pharma(A):- zinc(A,C), zinc(A,B), bond(B,C,du), dist(A,B,C,D), leq(D,1.23).
```

Comprehensibility

Logic programs are relatively comprehensible

Comprehensibility

Logic programs are relatively comprehensible

Comprehensibility is affected by:

- textual complexity
- predicate invention
- execution complexity

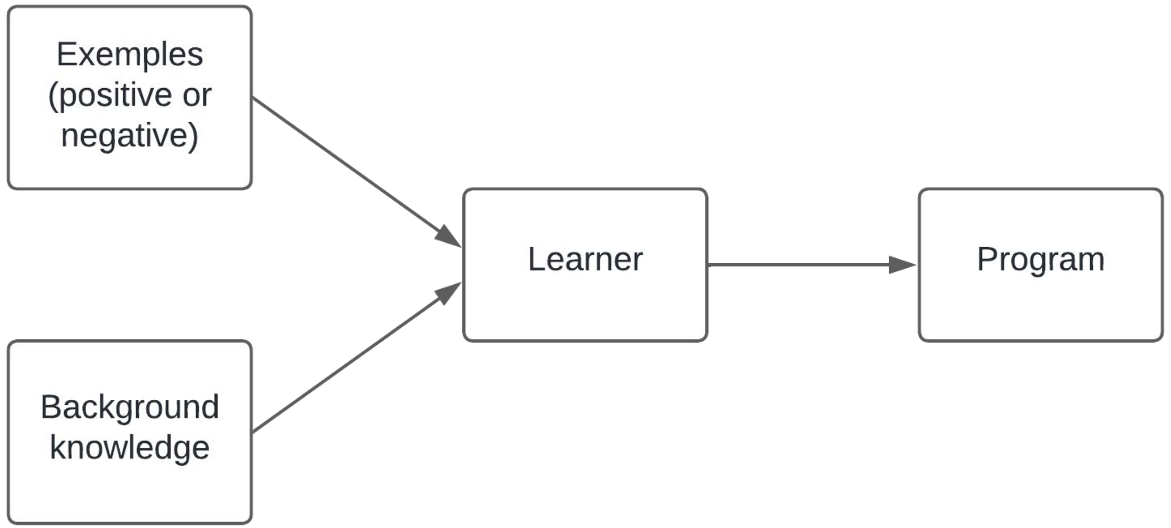
How does predicate invention affect human comprehensibility? Ute Schmid, Christina Zeller, Tarek Besold, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. Inductive Logic Programming, p. 52–67.

Beneficial and harmful explanatory machine learning, Lun Ai, Stephen Muggleton, Céline Hocquette, Mark Gromowski, and Ute Schmid, Machine Learning, 2021

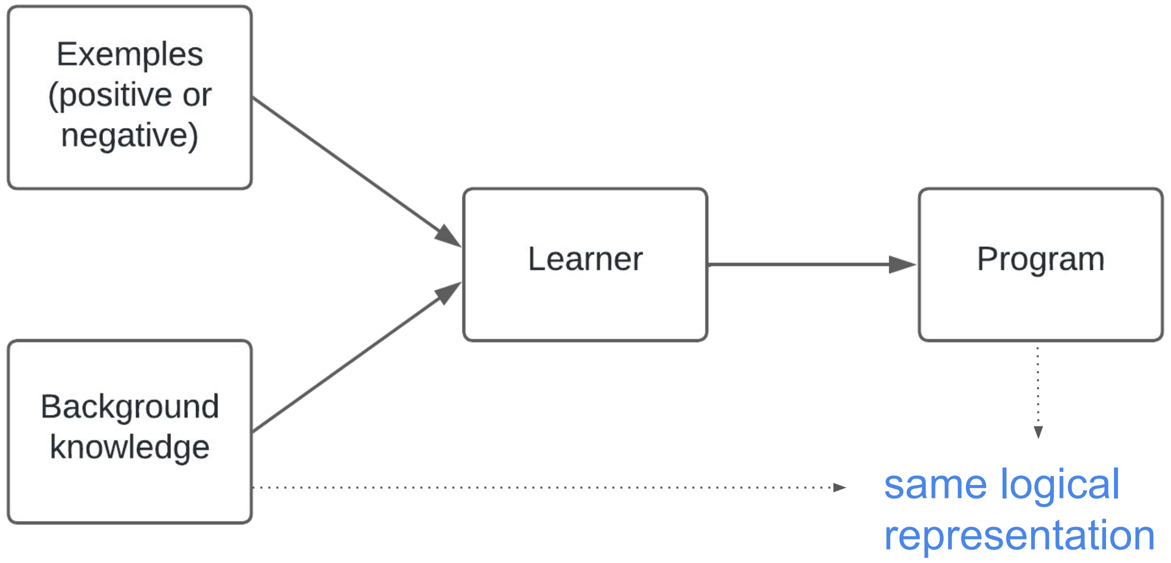
Lifelong learning

continuously learn through time

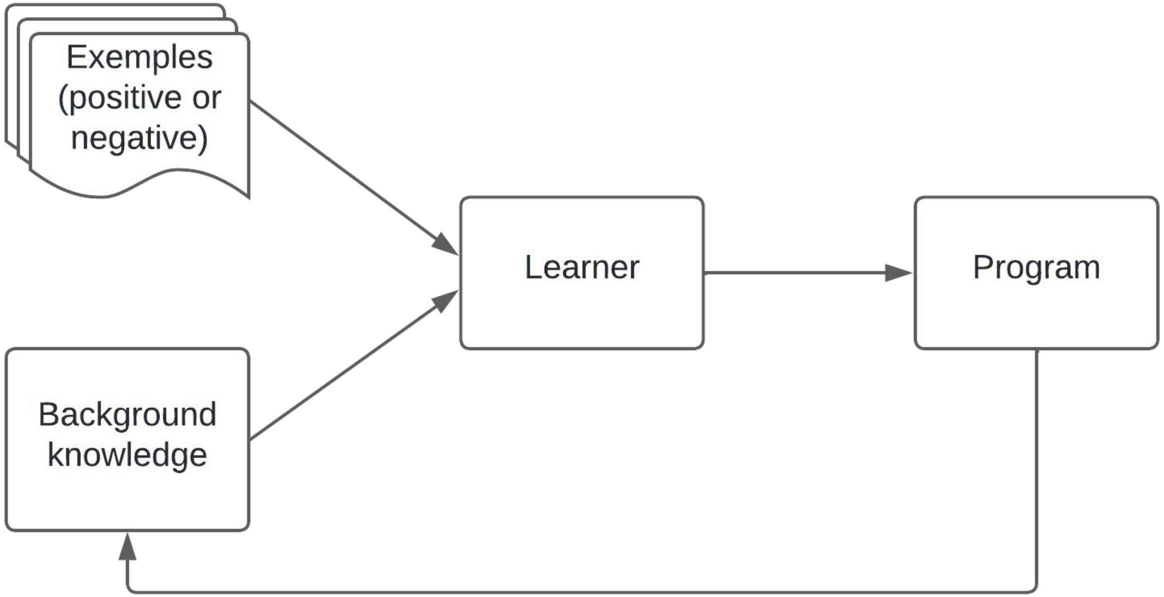
Inductive Logic Programming (ILP)
Single task



Inductive Logic Programming (ILP)



Lifelong learning: multiple tasks



Lifelong learning

Task 1

Input	Output
kutaisi	isiatuk
university	ytisrevinu

Lifelong learning

Task 1

Input	Output
kutaisi	isiatuk
university	ytisrevinu

```
reverse(A,B) ← empty(A),empty(B).
reverse(A,B) ← head(A,C),tail(A,D),reverse(D,E),append(E,C,B).
```

Lifelong learning

Task 2

Input	Output
kutaisi	isiatuk
university	ytisrevinu

Lifelong learning

Task 2

Input	Output
georgia	igqtikc
international	kpvgtpcvkqpcn

```
add2(A,B) ← map(inv1,A,B)
inv1(A,B) ← succ(A,C), succ(C,B).
```

Lifelong learning

Task 3

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

Lifelong learning

Task 3

Input	Output
inductive	gxkvewfpk
logic	ekiqn
programming	ipkooctiqtr

```
str_transformation(Input,Output) ← add2(Input,String), reverse(String,Output).
```

Lifelong learning

Limitation: the size of the search space is polynomial into the number of relations in the BK.

Saving too much BK can degrade learning performance.