universität
innsbruck

Smarter Features, Simpler Learning?

Georg Moser and Sarah Winkler

Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements
26 August 2019, Natal

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics

---

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models: naive Bayes, SVMs, random forests, . . . , neural networks

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models:	naive Bayes, SVMs, random forests, . . . , neural networks
features:	plain input, term walks, symbol/clause count, . . .

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

smarter features, simpler learning

Past/Current Work in Theorem Proving

models: naive Bayes, SVMs, random forests, . . . , neural networks
features: plain input, term walks, symbol/clause count, . . .

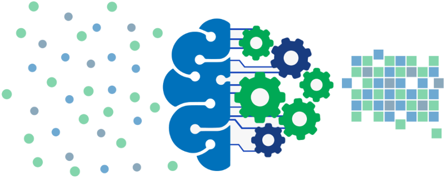
Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models:	naive Bayes, SVMs, random forests, . . . , neural networks
features:	plain input, term walks, symbol/clause count, . . .

theorem proving problem →  → strategy

Demyanova et al., Empirical Software Metrics for Benchmarking of Verification Tools, 2017.

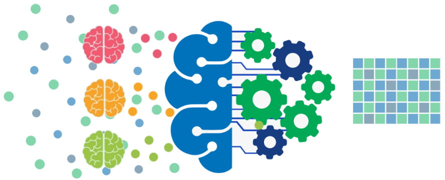
Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models: naive Bayes, SVMs, random forests, . . . , neural networks
features: plain input, term walks, symbol/clause count, . . .

theorem proving problem →



→ strategy

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

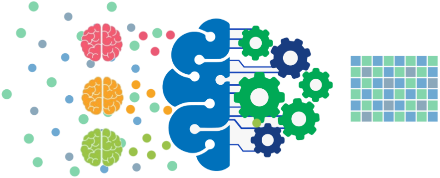
- strategy/tool are machine learned from program characteristics
- model
  occurrence count for 27 roles: pointers, loop bounds, counters, ...
- features:
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models: naive Bayes, SVMs, random forests, ..., neural networks

features: plain input, term walks, symbol/clause count, ...

theorem proving problem →



→ strategy

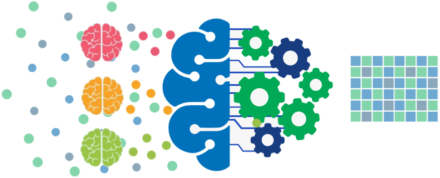
Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features: occurrence count for 3 types depending on iteration estimate
  - variable roles
  - loop patterns
  - control flow patterns
- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models: naive Bayes, SVMs, random forests, . . . , neural networks
features: plain input, term walks, symbol/clause count, . . .

theorem proving problem →  → strategy

Demyanova et al., Empirical Software Metrics for Benchmarking of Verification Tools, 2017.

Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- model: SVMs
- features:
  - variable roles
  - loop patterns
  - control flow patterns

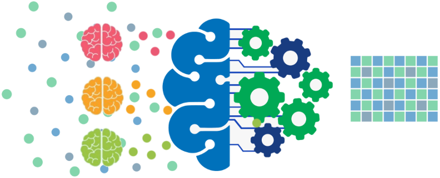
basic blocks, indegree, (recursive) calls

- would have won SV-COMP in 3 consecutive years

Past/Current Work in Theorem Proving

models: naive Bayes, SVMs, random forests, ..., neural networks
features: plain input, term walks, symbol/clause count, ...

theorem proving problem →



→ strategy

Demyanova et al., **Empirical Software Metrics for Benchmarking of Verification Tools**, 2017.

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:

Example

$$\mathsf{add}(0, x) \to x \quad (1) \qquad \mathsf{mul}(0, y) \to 0 \quad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \quad (4)$$

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \ldots, t_n) \to t_i$

Example

$$\mathsf{add}(0, x) \to x \quad (1) \qquad \mathsf{mul}(0, y) \to 0 \quad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \quad (4)$$

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \ldots, t_n) \to t_i$
  - i is decreasing for rule $f(\ldots, s(t_i), \ldots) \to C[f(\ldots, t_i, \ldots)]$

Example

$$\mathsf{add}(0, x) \to x \quad (1) \qquad \mathsf{mul}(0, y) \to 0 \quad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \quad (4)$$

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \ldots, t_n) \to t_i$
  - i is decreasing for rule $f(\ldots, s(t_i), \ldots) \to C[f(\ldots, t_i, \ldots)]$
  - recursive positions: recursive calls to same function symbol

Example

$$\mathsf{add}(0, x) \to x \quad (1) \qquad \mathsf{mul}(0, y) \to 0 \quad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \quad (4)$$

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \dots, t_n) \to t_i$
  - i is decreasing for rule $f(\dots, s(t_i), \dots) \to C[f(\dots, t_i, \dots)]$
  - recursive positions: recursive calls to same function symbol
  - pattern matching positions distinguish different constructors

Example

$$\mathsf{add}(0, x) \to x \quad (1) \qquad \mathsf{mul}(0, y) \to 0 \quad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \quad (4)$$

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \ldots, t_n) \to t_i$
  - i is decreasing for rule $f(\ldots, s(t_i), \ldots) \to C[f(\ldots, t_i, \ldots)]$
  - recursive positions: recursive calls to same function symbol
  - pattern matching positions distinguish different constructors
  - duplication positions contain variables which get duplicated

Example

$$\mathsf{add}(0, x) \to x \qquad (1) \qquad \mathsf{mul}(0, y) \to 0 \qquad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \qquad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \qquad (4)$$

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \dots, t_n) \to t_i$
  - i is decreasing for rule $f(\dots, s(t_i), \dots) \to C[f(\dots, t_i, \dots)]$
  - recursive positions: recursive calls to same function symbol
  - pattern matching positions distinguish different constructors
  - duplication positions contain variables which get duplicated
- loop patterns = recursion patterns: tiering and safe recursion

Example

$$\mathsf{add}(0, x) \to x \quad (1) \qquad \mathsf{mul}(0, y) \to 0 \quad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \quad (4)$$

Bellantoni and Cook, A new recursion-theoretic characterization of the polytime functions, 1992.

Possible Characteristics of Rewrite Systems

- variable roles = argument positions of function symbols:
  - i is projection argument in rule $f(t_1, \ldots, t_n) \to t_i$
  - i is decreasing for rule $f(\ldots, s(t_i), \ldots) \to C[f(\ldots, t_i, \ldots)]$
  - recursive positions: recursive calls to same function symbol
  - pattern matching positions distinguish different constructors
  - duplication positions contain variables which get duplicated
- loop patterns = recursion patterns: tiering and safe recursion
- control flow = call graph analysis:

  strongly connected components, in/out degree of nodes, edges between nodes of different root symbols, ...

Example

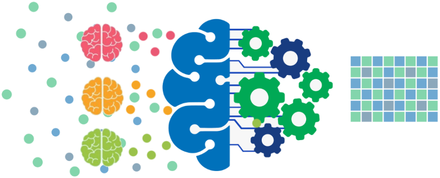
$$\mathsf{add}(0, x) \to x \qquad (1) \qquad\qquad \mathsf{mul}(0, y) \to 0 \qquad (3)$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \qquad (2) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y)) \qquad (4)$$

$$\begin{tikzcd}
(2) \arrow[r] \arrow[loop below] & (4) \arrow[loop below]
\end{tikzcd}$$

How about theorem proving in general?

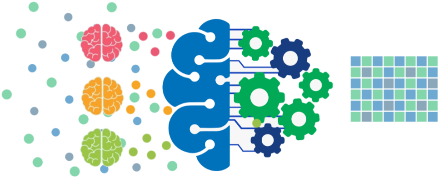
consider machine learning of strategies applied to a given problem:

theorem proving problem →  → strategy

How about theorem proving in general?

consider machine learning of strategies applied to a given problem:

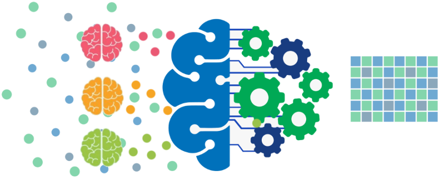
- can we preprocess characteristics from theorem proving problems which serve as useful features for learning?

theorem proving problem →  → strategy

How about theorem proving in general?

consider machine learning of strategies applied to a given problem:

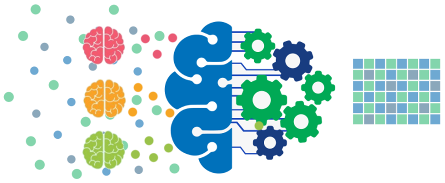
- can we preprocess characteristics from theorem proving problems which serve as useful features for learning?
- ... or better rely on neural networks discovering relevant characteristics by themselves?

theorem proving problem →  → strategy

How about theorem proving in general?

consider machine learning of strategies applied to a given problem:

- can we preprocess characteristics from theorem proving problems which serve as useful features for learning?
- ... or better rely on neural networks discovering relevant characteristics by themselves?
- how could such features look like?

theorem proving problem →  → strategy