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
Portfolio Solver for Software Verification Competition

- strategy/tool are machine learned from program characteristics
- **model**: SVMs
Portfolio Solver for Software Verification Competition

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

---

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

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, . . .

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, ...

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, . . .
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, ...

---

Possible Characteristics of Rewrite Systems

- **variable roles** = argument positions of function symbols:

Example

1. \( \text{add}(0, x) \rightarrow x \)
2. \( \text{add}(s(x), y) \rightarrow s(\text{add}(x, y)) \)
3. \( \text{mul}(0, y) \rightarrow 0 \)
4. \( \text{mul}(s(x), y) \rightarrow \text{add}(y, \text{mul}(x, y)) \)
Possible Characteristics of Rewrite Systems

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

Example

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x & (1) & \text{mul}(0, y) \rightarrow 0 & (3) \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) & (2) & \text{mul}(s(x), y) \rightarrow \text{add}(y, \text{mul}(x, y)) & (4)
\end{align*}
\]
Possible Characteristics of Rewrite Systems

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

**Example**

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x & (1) & \text{mul}(0, y) & \rightarrow 0 & (3) \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) & (2) & \text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y)) & (4)
\end{align*}
\]
Possible Characteristics of Rewrite Systems

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

**Example**

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x & (1) & \text{mul}(0, y) & \rightarrow 0 & (3) \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) & (2) & \text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y)) & (4)
\end{align*}
\]
Possible Characteristics of Rewrite Systems

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

---

**Example**

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x \quad (1) \\
\text{mul}(0, y) & \rightarrow 0 \quad (3) \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \quad (2) \\
\text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y)) \quad (4)
\end{align*}
\]
Possible Characteristics of Rewrite Systems

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

Example

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x & (1) \quad \text{mul}(0, y) & \rightarrow 0 & (3) \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) & (2) \quad \text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y)) & (4)
\end{align*}
\]
Possible Characteristics of Rewrite Systems

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

**Example**

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x \quad (1) \\
\text{mul}(0, y) & \rightarrow 0 \quad (3) \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \quad (2) \\
\text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y)) \quad (4)
\end{align*}
\]

Possible Characteristics of Rewrite Systems

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

1. \[
    \text{add}(0, x) \rightarrow x
\]
2. \[
    \text{add}(s(x), y) \rightarrow s(\text{add}(x, y))
\]
3. \[
    \text{mul}(0, y) \rightarrow 0
\]
4. \[
    \text{mul}(s(x), y) \rightarrow \text{add}(y, \text{mul}(x, y))
\]

\[
\begin{array}{c}
\text{(2)} \\
\cup
\end{array}
\quad
\begin{array}{c}
\text{(4)} \\
\cup
\end{array}
\]
How about theorem proving in general?

consider machine learning of strategies applied to a given problem:

\[\text{theo} \text{rem proving problem} \rightarrow \text{strategy}\]
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?

\[
\text{theorem proving problem} \rightarrow \text{strategy}
\]
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 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?