The promise and perils of using machine learning when engineering software.

Yuriy Brun

Machine learning today

The promise and perils of using machine learning when engineering software.

Yuriy Brun

Modern software uses machine learning to influence critical decisions.

Software can make bad decisions. Software can discriminate!
Machine learning has great promise, but with that promise, come risks.

Today’s goal:
Identifying and addressing the risks
A simpler problem: Automated program repair

- Tweak the program
- Check if tests pass
- If not, repeat

Program repair techniques

- Evaluated 4 techniques
  - GenProg
  - Par
  - TriAutoRepair
  - SimFix
- Measured patch quality
- Measured what affects patch quality

Program repair techniques

APR is a form of machine learning

- first, many techniques rely on ML to learn
  - where to edit the code
  - how to edit the code
  - how to decide which patches are good
- second, the underlying problem is learning a function (program) using training data (tests)

How well does APR work?

Quality vs. quantity

When applied to real-world Java code, APR produces patches for 10.6-19.0% of the defects
Quality vs. quantity

Potential problem: Overfitting
APR uses a set of tests to guide repair. Tests are inherently partial. No way APR can know if a patch captures intended behavioral constraints.

Does APR at least improve things a bit?

<table>
<thead>
<tr>
<th>Technique</th>
<th>Minimum</th>
<th>Mean</th>
<th>Median</th>
<th>Maximum</th>
<th>100%-quality patches</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenProg</td>
<td>64.8%</td>
<td>94.7%</td>
<td>98.4%</td>
<td>100.0%</td>
<td>24.3%</td>
</tr>
<tr>
<td>Par</td>
<td>64.8%</td>
<td>96.1%</td>
<td>98.5%</td>
<td>100.0%</td>
<td>33.8%</td>
</tr>
<tr>
<td>SimFix</td>
<td>65.0%</td>
<td>96.3%</td>
<td>98.9%</td>
<td>100.0%</td>
<td>46.1%</td>
</tr>
<tr>
<td>TripAutoRepair</td>
<td>64.8%</td>
<td>96.4%</td>
<td>98.4%</td>
<td>100.0%</td>
<td>19.5%</td>
</tr>
</tbody>
</table>

Less than half (14-46%) of the patches are correct

Does APR at least improve things a bit?

<table>
<thead>
<tr>
<th>Technique</th>
<th>Minimum</th>
<th>Mean</th>
<th>Median</th>
<th>Maximum</th>
<th>Improvement</th>
<th>No change</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenProg</td>
<td>−36.9%</td>
<td>−1.7%</td>
<td>0.0%</td>
<td>2.6%</td>
<td>−36.9%</td>
<td>−1.7%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Par</td>
<td>−26.9%</td>
<td>−2.8%</td>
<td>0.0%</td>
<td>1.5%</td>
<td>−26.9%</td>
<td>−2.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>SimFix</td>
<td>−24.9%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>35.0%</td>
<td>−24.9%</td>
<td>0.2%</td>
<td>0.0%</td>
</tr>
<tr>
<td>TripAutoRepair</td>
<td>−26.9%</td>
<td>−2.1%</td>
<td>0.0%</td>
<td>2.8%</td>
<td>−26.9%</td>
<td>−2.1%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair

Smith, et al., Is the Cure Worse That the Disease? Overating in Automated Program Repair, ESEC/FSE 2015.
Takeaway: Tests are an imperfect oracle, so APR suffers, producing low-quality patches.

Can we find a domain with better oracles?

**Proposition:** Use APR-style technology to synthesize proofs

**Step 1:** Build a predictive model

**Step 2:** Guide search with the model

---

**Interactive theorem provers for formal verification**

Formal verification comes with a built-in oracle: The theorem prover

**Industrial impact of theorem proving**

**Prohibitively difficult**

Verified software requires a lot of time and a lot of proofs in proportion to code

- Proof is about 8 times bigger than the compiler code
- 3 person years of work

Virtually all software that ships today is unverified.
Proposal: Use APR-style technology to synthesize proofs

Step 1: Build a predictive model

Step 2: Guide search with the model

How to learn a predictive model

Step 1: Build a predictive model

Step 2: Guide search with the model

Proposal: Use APR-style technology to synthesize proofs

Step 2: Guide search with the model
TacTok (OOPSLA’20)

TacTok models partial proof and the current proof state, together

CoqGym Dataset

- 123 open-source software projects in Coq
- 70,856 theorems
- Broken down into 96 projects (57,719 proofs) for training and 27 projects (13,137 theorems) for testing

https://github.com/princeton-vl/CoqGym

TacTok vs. ASTactic vs. SeqOnly

TacTok vs. ASTactic vs. CoqHammer

CoqHammer

7,500 (69.6%) unproven theorems

Vary:
- proof tactic and token depth
- learning rate
- embedding size
- number of layers
- training order
- access to proof state, partial proof, Gallina proof term

Diva (ICSE’22)

2 key observations:
- Machine learning is often noisy
- Theorem prover serves as an oracle to turn that noise into signal.
Diva vs. state-of-the-art

Diversity inherent in ML increases the proving power 68%-77% over prior search-based synthesis tools, and 27% over CoqHammer.

https://github.com/LASER-UMASS/Diva/

Fully Automated Formal Verification

Machine learning and meta-heuristic search can fully automate some bug-repair and formal verification.

While APR underperforms because it is driven by an unreliable oracle, formal verification is a killer app for APR because the theorem prover provides a reliable oracle.

...let’s talk about a different peril of machine learning that verification might help with.

Part II
Software discrimination

Data-driven systems can exhibit undesirable properties. Can we build systems to be safe and fair?

Testing systems for bias

Themis
automated test-suite generator

How much does my software discriminate with respect to ...?

Does my software discriminate more than 10% of the time, and against what?

http://fairness.cs.umass.edu

Debugging bias

Can we verify systems to be safe and fair?

How would that work?

User specifies a definition of safe or fair behavior.

Train classifiers, selects one to satisfy fairness, verify safety on held-out suite.

Example scenario:

Suppose a university wants to train a model to predict student success from entrance exam scores, while ensuring the model is fair: roughly the same fraction of men and women are predicted to be successful. (This is called Disparate Impact.)
Example scenario:

One source of ML bias comes from deploying a model on data that is fundamentally different from the data the model was trained on.

What if software is deployed on data fundamentally different from training data?
Machine learning can result in unexpected, unintended behavior. But machine learning can be leveraged to produce verified safe and fair models, avoiding such behavior.