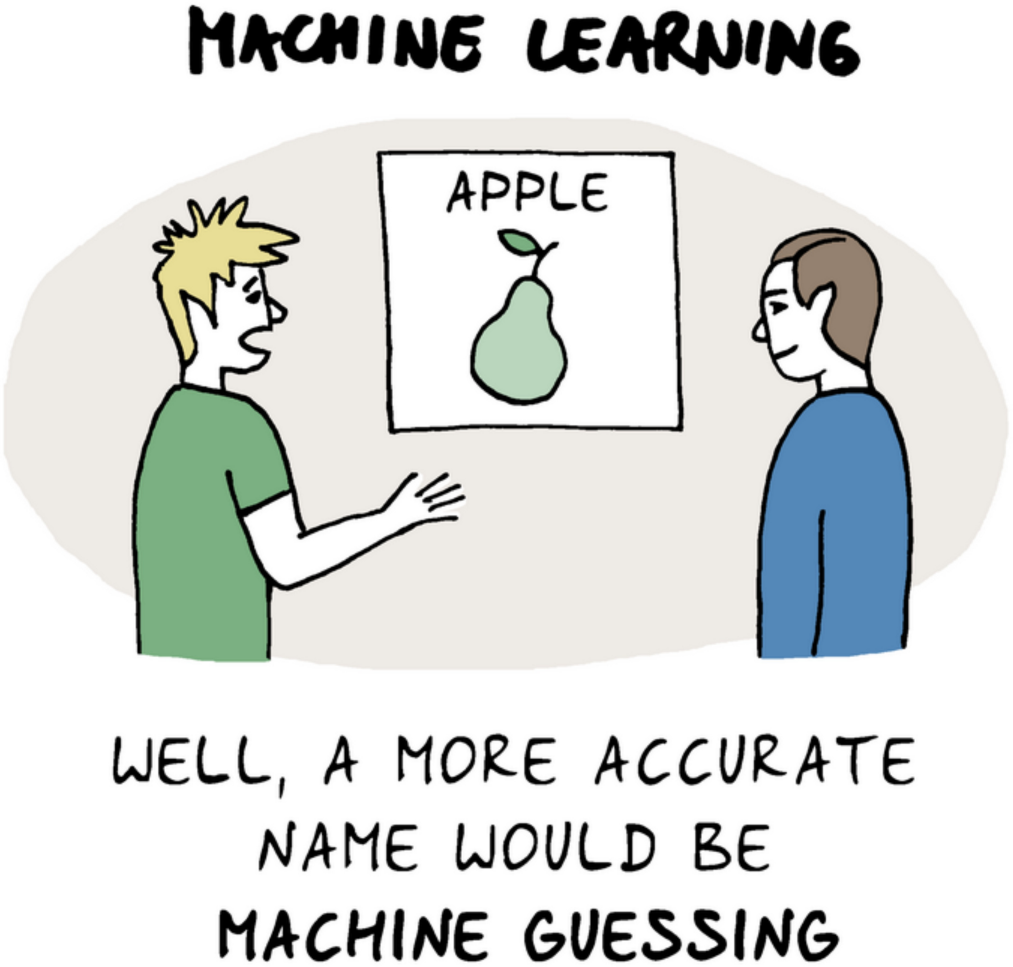
AI & Pre- and Post-Conditions



One-Slide Summary

- Many PL techniques require predicates describing program behavior (pre- and post-conditions, invariants, refinement types, etc.). LLMs can statically generate candidate invariants from program source code and comments.
- We can also use PL to address problems in LLMs. For example, axiomatic semantics can be used to predict hallucinations and increase trust. “Proving” LLMs correct is a hot research topic that is not yet solved; many proposed solutions do not scale.

Reprise: Question Set #4

Wu et al.'s *Lemur: Integrating*

- Read Section 2 and 3.0 and discuss:
- Does Stable → Invariant? Invariant → Stable?
- When would the verifier return Unknown?
- Is an LLM used for $O_{propose}$, O_{repair}, neither, or both?
- In the Propose rule in Figure 1, what do we know about $\mathcal{V}(\mathcal{P}, \mathcal{A}, q)$? What do we suspect?
  - Note: q is not a typo.
- How do they prove Theorem 3.1 and 3.2?

Does It Work? Quality

- "We found that the LLM-based oracles can produce surprisingly insightful loop invariants that are difficult for conventional formal methods to synthesize. While predicate-abstraction-based techniques typically generate predicates that involve only the operators and values in the program and follow a particular template, LLM is not constrained by these limitations. For example, for the program in Fig. 4, GPT-4 can consistently generate x%4==0 as the loop invariant although the modulo operator is not present in the program."
  - How does this compare to Daikon? DIG? Newton from SLAM?
- "There are also several cases where the LLM generates disjunctive invariants that precisely characterize the behavior of the loops."
  - How does this compare to Daikon? DIG? Newton from SLAM?

Does It Work? Quantity

- They outperform ESBMC (symbolic: bounded model checker), Code2Inv (neural: reinforcement learning), Uautomizer (symbolic: program analysis, prior winner of SV-COMP)

Configurations	Solved	Time	# proposal
Code2Inv	92	–	20
ESBMC	68	0.34	0
LEMUR	107	24.9	4.7

(a) The Code2Inv benchmarks.

Configurations	Solved	Time	# proposals
UAUTOMIZER	0	–	0
ESBMC	0	–	0
LEMUR	26	140.7	9.1

(b) The 50 SV-COMP benchmarks.

- They are also the first approach to include learning that can handle 2+ loops

Invariants and Assertions

- Many formal techniques require knowing a predicate that describes program behavior
  - Example Predicate: `x <= 5`
  - Use: VCGen while$_{Inv}$ b do c
  - Use: Axiomatic Inv1 => WP(c, Inv2)
  - Use: Dependent Type Refinements { x | Inv }
  - Use: Synthesis (write a function that Inv)
- Invariants, pre- and post-conditions and refinements all need such predicates

Invariant Detection

- Classic invariant detection algorithms (Daikon, DIG, etc.) are dynamic analyses: they require that you can compile and run the program and that you have indicate workloads

- What if we could generate candidate invariants just from the "natural language" of the program (e.g., comments, code)?
  - This would be a static analysis (and thus more broadly applicable)

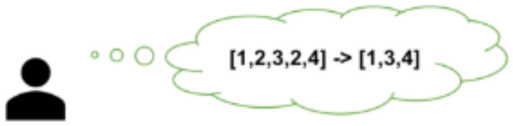
Reading and Understanding

- Once again we will discuss recent papers in class, looking at direct descriptions and also less-obvious implications
- This time the papers were assigned readings, so you are all already familiar with them
  - If this isn't true, we may have a reading quiz
  - If it turns out to be true, we may skip the quiz
- I will call on you to read short passages out loud and answer questions

Let's Explore Together

Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?

MADELINE ENDRES*, University of Michigan, USA
SARAH FAKHOURY, Microsoft Research, USA
SAIKAT CHAKRABORTY, Microsoft Research, USA
SHUVENDU K. LAHIRI, Microsoft Research, USA



(a) Programmer intent for a function that removes all instances of numbers that have duplicates from a list.

```
1 def remove_duplicates(numbers: List[int]):
2   """ From a list of integers, remove all elements that occur more than
        once. Keep order of elements left the same as in the input """
```

(b) *Ambiguous* natural language specification: it does not specify if all copies or all but one copy of a duplicated element should be removed. In this case, the programmer intends the former.

`1 assert len(set(numbers)) == len(set(return_list))`	✗
`1 assert all(numbers.count(i) == 1 for i in return_list)`	✓

(c) Postconditions generated by GPT-4. Note that while both could be correct with a literal reading of the natural language specification, only the second one is correct with respect to developer intent.

Problem Framing

- Which two properties are implicitly claimed to be necessary for a good solution?

The "emergent abilities" of Large Language Models (LLMs) have the potential to facilitate the translation of natural language intent to programmatically checkable assertions. However, it is unclear if LLMs can correctly translate informal natural language specifications into formal specifications that match programmer intent. Additionally, it is unclear if such translation could be useful in practice.

In this paper, we describe *nl2postcond*, the problem of leveraging LLMs for transforming informal natural language to formal method postconditions, expressed as program assertions. We introduce and validate metrics to measure and compare different *nl2postcond* approaches, using the correctness and *discriminative power* of generated postconditions. We then use qualitative and quantitative methods to assess the quality of *nl2postcond* postconditions, finding that they are generally correct and able to discriminate incorrect code. Finally, we find that *nl2postcond* via LLMs has the potential to be helpful in practice; *nl2postcond* generated postconditions were able to catch 64 real-world historical bugs from *Defects4J*.

- Many researchers use the "Heilmeier Catechism" to structure papers and proposals

Concepts

solution r, optimizing for a number of outcomes. First, the prompts should work with *chat-based models*, and the generated postconditions should be *symbolic* (e.g., not point-wise tests), directly executable, and side-effect free. Also, the prompt should encourage the LLM to produce expressions

a few specifications that are used to constrain AI generated code suggestions. However, we also provide a prompt variant that includes the reference code r along with *nl*. This allows us to assess if natural language alone can be as effective as code in conveying programming intent to an LLM.

function. When using the `base` prompt, LLMs have a tendency to construct complex postconditions, often approaching a fully functional implementation of the problem. While useful, we observe these postconditions are likely to be incorrect. To illustrate this, Fig. 4 compares postconditions

- Which quote is about each of: recall, overfitting, type systems, neurosymbolic?

Science Discussion

possible. Several prompt iterations were considered until we observed satisfactory performance on a subset of example problems, though we acknowledge further prompt tuning may result in different outcomes. Figure 3 outlines our prompt template. This template shows four possible

- What is HARKing? What is False Discovery Rate?
- Give one negative view of this aspect of the paper. Give one positive or mitigating view of this aspect of the paper.

Correctness vs. Test Cases

Model	Prompt	Prompt has: NL Only=✗ ref code=✓	Accept @ 1	Accept @ 5	Accept @ 10	x/164 correct
GPT-3.5	base	✗	0.46	0.80	0.87	143
GPT-3.5	base	✓	0.49	0.81	**0.88**	**145**
GPT-3.5	simple	✗	0.55	**0.82**	0.87	143
GPT-3.5	simple	✓	**0.56**	**0.82**	**0.88**	144
GPT-4	base	✗	0.63	0.83	0.88	144
GPT-4	base	✓	0.71	0.89	0.91	150
GPT-4	simple	✗	**0.77**	**0.94**	**0.96**	**158**
GPT-4	simple	✓	0.76	0.92	**0.96**	157
StarChat	base	✗	0.21	0.61	0.82	134
StarChat	base	✓	0.20	0.59	0.77	126
StarChat	simple	✗	**0.25**	**0.69**	0.85	139
StarChat	simple	✓	0.23	0.67	**0.86**	**141**

- What is ablation? What is Accept@1? (They never actually define either before use.)
- "Did they win?"

“Completeness”

Category	Example Postconditon	% Prevalent	Avg. Bug-complete-score (*Natural*/*All*)
Type Check	`isinstance(return_val, int)`	47.4	*0.14 / 0.27*
Format Check	`return_val.startswith("ab")`	11.2	0.43 / 0.57
Arithmetic Bounds	`return_val >= 0`	30.8	0.23 / 0.34
Arithmetic Equality	`return_val[0] == 2 * input_val`	17.5	**0.82** / **0.89**
Container Property	`len(return_val) > len(input_val)`	27.0	0.45 / 0.57
Element Property	`return_val[0] % 2 == 0`	12.6	0.39 / 0.53
Forall-Element Property	`all(ch.isalpha() for ch in return_val)`	8.3	0.23 / 0.44
Implication	`(return_val==False) if 'A'not in string`	12.7	0.58 / 0.64
Null Check	`return_val is not None`	4.4	0.40 / 0.50
Average			0.32 / 0.46

RQ1 summary: Postcondition Completeness on *EvalPlus*

We find that for the benchmark *EvalPlus*, *nl2postcond* postconditions generated by `GPT-3.5` and `GPT-4` can meaningfully capture program intent especially when using our `base` prompt: the average correct postcondition generated by these models can discriminate three-quarters of unique buggy code mutants depending on the prompt variation.

- What does “can discriminate X% of unique buggy code mutants” mean?
- Could Daikon/DIG make these postconditions? Did they?

Qualitative Analysis

We did not observe a significant relation between postcondition type and correctness. However, we do observe significant differences in bug-completeness across categories. For example, postconditions labeled as `Type Checks`, i.e. specifications enforcing the type of the return value, were the weakest, only killing 27% of bugs on average. This difference was particularly pronounced for *natural bugs* (see Sections 2.1 and 3.1.4), where `Type Checkers` only killed 14% of bugs on average. Interestingly, `Type Checks` are also the most prevalent category, indicating LLM preference towards generating such constraints. Low completeness scores indicate that, for the studied dataset,

On the other hand, `Arithmetic Equality` checks, i.e. specifications that assert that parts of the return value must be equivalent to another expression, provide a strong postcondition. On average, this category of postcondition kills 89% of all bugs and appears in 17.5% of labeled postconditions.

- Why might type-checking postconditions be so weak?
  - "Is Grad PL a lie?"

Bug-Finding via Postconditions

Table 4. Table containing our *Defects4J* results for postconditions generated for 840 methods across 525 historical bugs. We report the likelihood of generated postconditions to compile, and the `accept@k` likelihood that they pass all tests when instrumenting the fixed function (*test-set correct* columns). *# distinguishable bugs* is the number of bugs for which at least one generated postcondition was discriminating (see Section 4.1.2).

Model	Prompt has: NL Only = ✗ buggy code = ✓	Compiles			Test-set correct			Number distinguishable bugs
		@1	@5	@10	@1	@5	@10	
GPT-4	✗	0.65	0.86	0.89	0.32	0.57	0.66	35
GPT-4	✓	**0.73**	**0.90**	**0.93**	**0.39**	**0.66**	**0.75**	**47**
StarChat	✗	0.25	0.68	0.83	0.11	0.38	0.55	19
StarChat	✓	**0.29**	**0.72**	**0.84**	**0.12**	**0.39**	**0.56**	**24**

- Is statically finding 19-47 of 525 bugs “good”?

History

- This founder of the Persian Achaemenid Empire conquered much of West and Central Asia (~550 BCE). He gained respect for his policy of respecting the customs and religions of those he conquered. He is generally recognized for achievements in human rights (his cylinder is sometimes described as the first human rights charter), politics (including bureaucracy like a post office), and military strategy (including an elite heavy infantry of “Immortals”).

Film

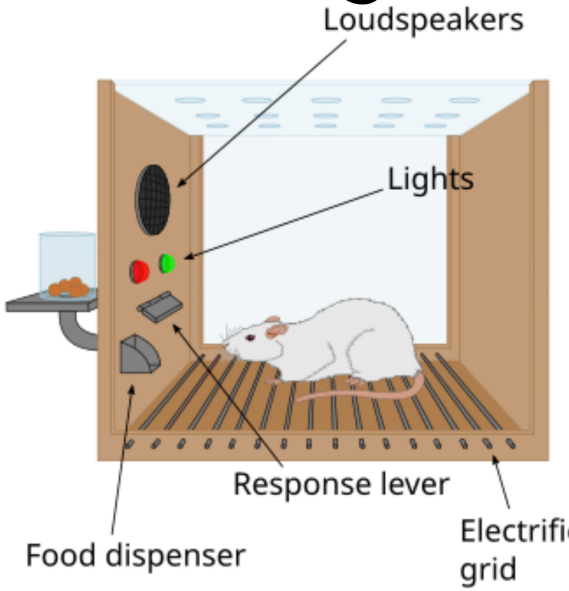
- This English film directory is widely viewed as one of the most influential. He is particularly known for his mastery of suspense. The Birds, Rope, North by Northwest, Rear Window and Psycho are all associated with him.

Fine Arts

- This private performing arts conservatory in New York City is a prestigious academy of dance, drama, and especially music. It has one of the lowest acceptance rates in the United States. Alumni include Robin Williams, Yo-Yo Ma, Philip Glass, and Wynton Marsalis.

Psychology

- This American Psychologist is associated with behaviorism (~1950). The theory equates behavior with the response to a stimulus. It included a focus on operant conditioning and has largely been dropped in favor of cognitive psychology.



AI for PL vs. PL for AI

- In conferences it is increasingly common to see some papers use AI to solve PL problems and a few papers use PL to solve AI problems
- We have predominantly consider using LLMs to solve programming problems, but our final paper uses PL techniques applied to LLMS “as programs”
  - “If you're worried about LLMs giving wrong answers, why not check the LLM for correctness with formal PL techniques?”

Inferring Data Preconditions from Deep Learning Models for Trustworthy Prediction in Deployment

Shibbir Ahmed
Dept. of Computer Science
Iowa State University
Ames, IA, USA
shibbir@iastate.edu

Hongyang Gao
Dept. of Computer Science
Iowa State University
Ames, IA, USA
hygao@iastate.edu

Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, IA, USA
hridesh@iastate.edu

In this work, we propose a novel technique that uses rules derived from neural network computations to infer data preconditions for a DNN model to determine the trustworthiness of its predictions. Our approach, *DeepInfer* involves introducing a novel abstraction for a trained DNN model that enables weakest precondition reasoning using Dijkstra's Predicate Transformer Semantics. By deriving rules over the inductive type of neural network abstract representation, we can overcome the matrix dimensionality issues that arise from the backward non-linear computation from the output layer to the input layer. We utilize the weakest precondition computation using rules of each kind of activation function to compute layer-wise precondition from the given postcondition on the final output of a deep neural network. We extensively evaluated *DeepInfer* on 29 real-

The Cunning Plan (1/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

The Cunning Plan (2/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

The Cunning Plan (3/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

```
def bar(z):
  assert(___)
  c = 5 - z*z
  assert(___)
  return c
def foo(y):
  assert(___)
  b = bar(len(y))-1
  assert(___)
  return b
def main(x):
  assert(___)
  a = foo(x+"w")
  assert(a>0) # post
  return a
```

The Cunning Plan (4/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

- Starting from the postcondition, what are the first three we compute?

```
def bar(z):
  assert(____)
  c = 5 - z*z
  assert(____)
  return c
def foo(y):
  assert(____)
  b = bar(len(y))-1
  assert(____)
  return b
def main(x):
  assert(____)
  a = foo(x+"w")
  assert(a>0) # post
  return a
```

3rd

2nd

1st

The Cunning Plan (5/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

```
def bar(z):
  assert(5-z^2>1)
  c = 5 - z*z
  assert(c>1)
  return c
def foo(y):
  assert(____)
  b = bar(len(y))-1
  assert(b>0)
  return b
def main(x):
  assert(____)
  a = foo(x+"w")
  assert(a>0) # post
  return a
```

The Cunning Plan (6/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

- Continuing down from the top, what are final conditions?

```
def bar(z):
  assert(5-z^2>1)
  c = 5 - z*z
  assert(c>1)
  return c
def foo(y):
  assert(____)
  b = bar(len(y))-1
  assert(b>0)
  return b
def main(x):
  assert(____)
  a = foo(x+"w")
  assert(a>0) # post
  return a
```

The Cunning Plan (7/7)

correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N + 1$) in a DNN, our approach then uses the preconditions of the $N + 1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

- Example data precondition:

  $5 - (\text{len}(x)+1)^2 > 1$

```
def bar(z):
  assert(5-z^2>1)
  c = 5 - z*z
  assert(c>1)
  return c
def foo(y):
  assert(5-len^2(y)>1)
  b = bar(len(y))-1
  assert(b>0)
  return b
def main(x):
  assert(5-len^2(x+"w")>1)
  a = foo(x+"w")
  assert(a>0) # post
  return a
```

Figure 4: Rules for computing wp

- Recall that α is "also" *wp* in their paper

$$\text{(WP)}\quad \frac{wp(N_0, \delta') = \delta'' \quad \delta' = wp(N_1, \delta)}{wp(N_0.N_1, \delta) = \delta''}$$

$$\frac{\alpha(\delta_0, \beta(a(f(\overline{x})))) = \delta_0' \quad \alpha(\delta_1, \beta(a(f(\overline{x})))) = \delta_1'}{\alpha(\delta_0 \wedge \delta_1, \beta(a(f(\overline{x})))) = \delta_0' \wedge \delta_1'}$$

- Fill in this totally unrelated wp rule:

$$\frac{wp(\_\_\_, bafx) = \_\_\_ \quad wp(\_\_\_, \_\_\_) = \_\_\_}{wp(\text{if } * \text{ then } s0 \text{ else } s1, bafx) = p0 \wedge p1}$$

Experiments

- Which of these aspects are good? Bad?

4.1.1 Benchmark. We have gathered four canonical real-world datasets from Kaggle competitions [41]. The train and test datasets

Dataset	# Features	Model	Source	# Layers	# Neurons
Pima Diabetes [61]	8	PD1	Kaggle	3	221
		PD2	Kaggle	3	221
		PD3	Kaggle	3	221

puting power or memory, it is crucial to ensure that models are suitable for deployment in safety-critical scenarios to prevent accidents or mitigate risks. For instance, a self-driving Uber car struck and killed a woman in March 2018 as an investigation [3] revealed that the model couldn't correctly predict her path and it needed to brake just 1.3 seconds before it struck her. Therefore, it is important to measure the runtime of such techniques.

Time (sec)
0.67
0.66
0.65
0.65
0.86
0.83
0.97
0.87
3.42
3.48
3.78
3.78
3.39
3.48
3.40

Did It Work?

- “DeepInfer implies that data precondition violations and Incorrect model prediction are highly correlated (0.88) between prediction ground truth and violation. Also, the precondition satisfaction and correct model prediction are strongly correlated (0.98).”
- “DeepInfer effectively implies the correct and incorrect prediction of higher accuracy models with recall (0.98) and F-1 score (0.84), compared to SelfChecker with recall (0.59) and F-1 score (0.52).”

Time Permitting

- In-Class HW6 Discussion

Homework

- HW6
- Reading Quiz?