Analyzing Software using Deep Learning

Robustness and Explainability

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Motivation

- **Neural models of code:**
  Hard to understand
  - Why do we get this prediction?
  - What properties of the code does the model learn from?
  - Does slightly modifying the code lead to a different prediction?
  - How to explain a prediction to a user?

Robustness

- **Want: Irrelevant changes should not affect model's predictions**
  - Slightly modified identifier names
  - Semantically equivalent code
- **Lack of robustness causes**
  - Surprising predictions $\rightarrow$ Unsatisfied users
  - Easy to circumvent models
    - Important for vulnerability detection models

Explainability

- **Want: Understand what causes a specific prediction**
  - A.k.a. local explanations
  - Crucial for user acceptance
- **Want: Understand how the model works in general**
  - A.k.a. global explanations
  - Import to avoid coincidental accuracy
  - Helps improve future models

Overview

- **Robustness** ←
- **Explaining specific predictions**
- **Explaining entire models**

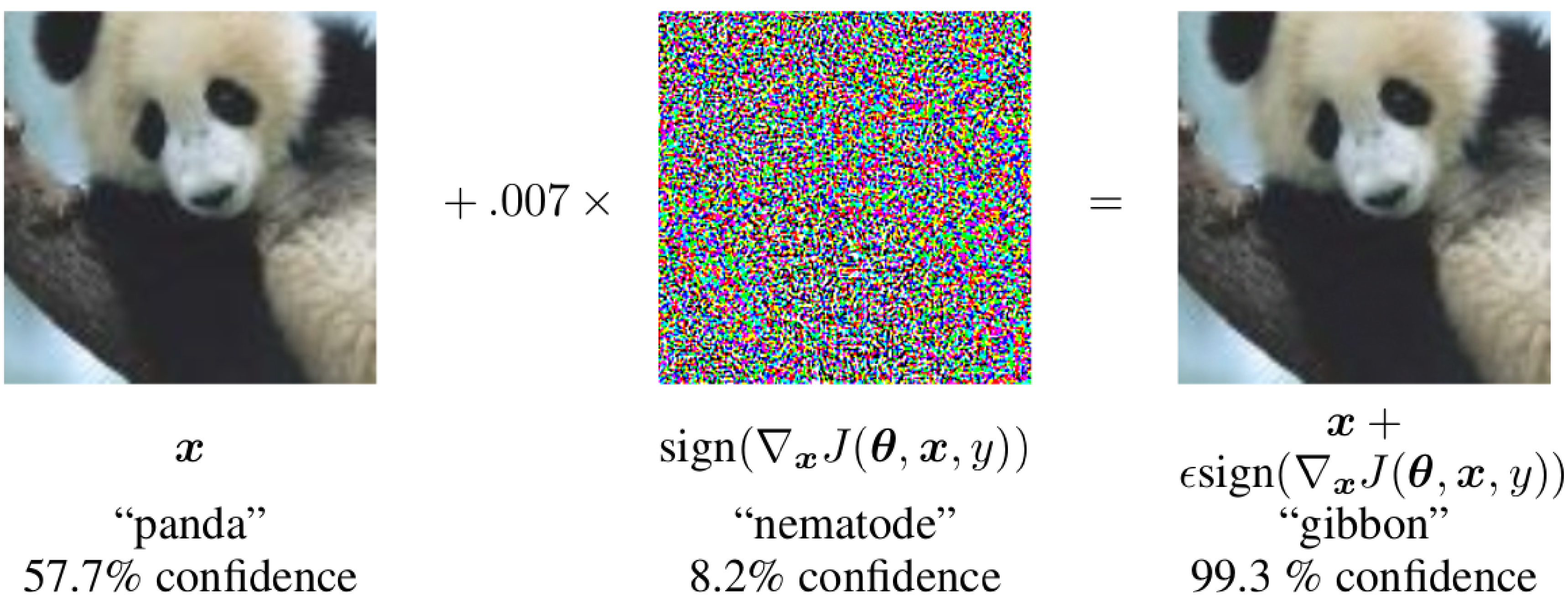
Recommended papers:

- "Adversarial Examples for Models of Code", Yefet et al., 2020
- "Counterfactual Explanations for Models of Code", Cito et al., 2022
- "Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code", Paltenghi et al., 2021

Adversarial Examples

Neural models:

Vulnerable to adversarial examples



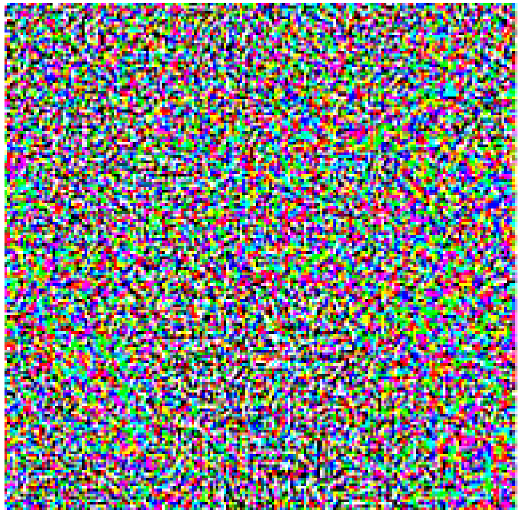
Explaining and harnessing adversarial examples, Goodfellow et al., 2014

Adversarial Code Examples

```
void f1(int[] array){
  boolean swapped = true;
  for (int i = 0;
    i < array.length && swapped; i++){
    swapped = false;
    for (int j = 0;
    j < array.length-1-i; j++) {
      if (array[j] > array[j+1]) {
        int temp = array[j];
        array[j] = array[j+1];
        array[j+1]= temp;
        swapped = true;
      }
    }
  }
}
```

Prediction: sort (98.54%)

+



= ?

Kinds of Attacks

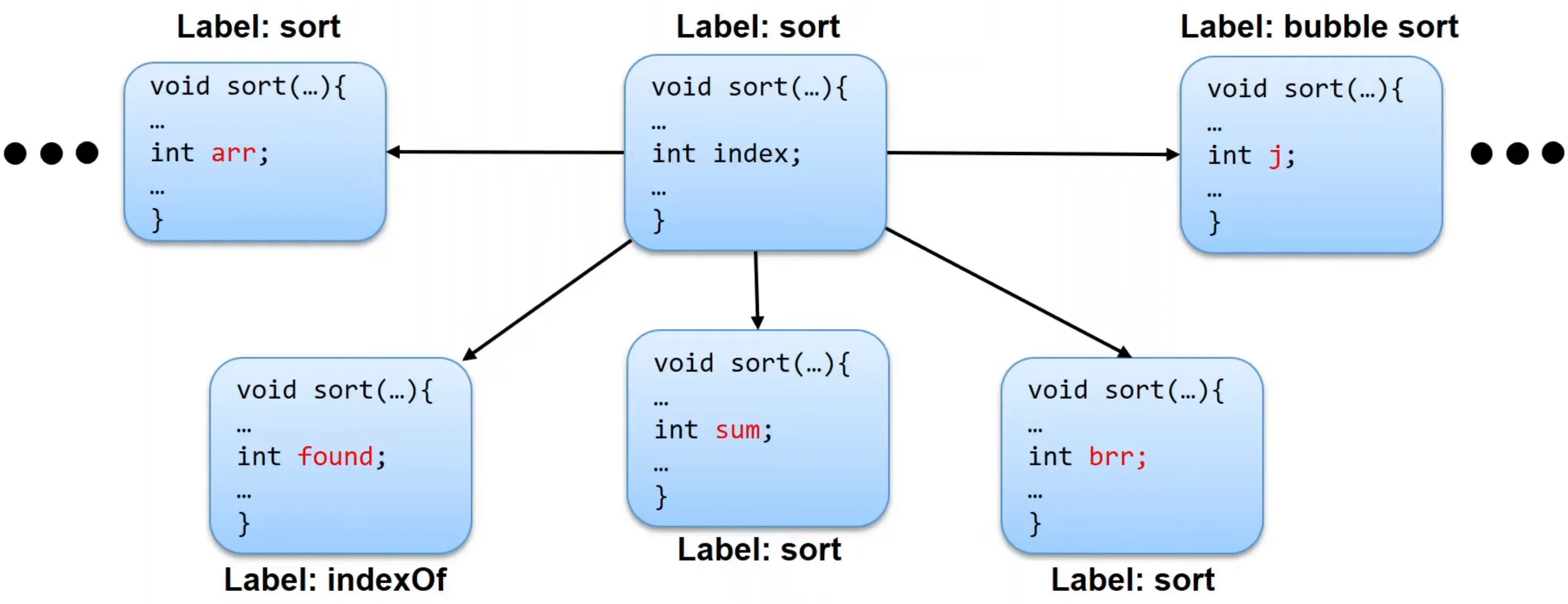
- **Given: Program p with correct label l**
- **Non-targeted attack**
  - Find “noise” to be added to p that yields a label $l' \neq l$
- **Targeted attack**
  - Find “noise” to be added to p that yields a specific label $l_{target} \neq l$

Adding Noise

- **How to add noise to programs?**
- **Semantics-preserving transformations**
  - Rename variables
  - Insert dead code
  - Remove dead code
  - Re-order independent statements
  - Modify content of comments
  - etc.

Space of Program Variants

How to hit a specific target label?



Gradient-Based Exploration

- **Explore input space via gradient-based exploration**
- **Similar to model training, but**
  - Model weights are fixed
  - Output is fixed to l_{target}
  - Update the input vector of one variable name

Examples

Robustness of Code2vec:

(predicts names of methods)

```
boolean f(Object target) {
  for (Object elem: this.elements) {
    if (elem.equals(target)) {
      return true;
    }
  }
  return false;
}
```

contains | 90.93%

```
boolean f(Object mist) {
  for (Object upperhexdigits: this.elements) {
    if (upperhexdigits.equals(mist)) {
      return true;
    }
  }
  return false;
}
```

escape | 99.97%

```
boolean f(Object target) {
  for (Object musicservice: this.elements) {
    if (musicservice.equals(target)) {
      return true;
    }
  }
  return false;
}
```

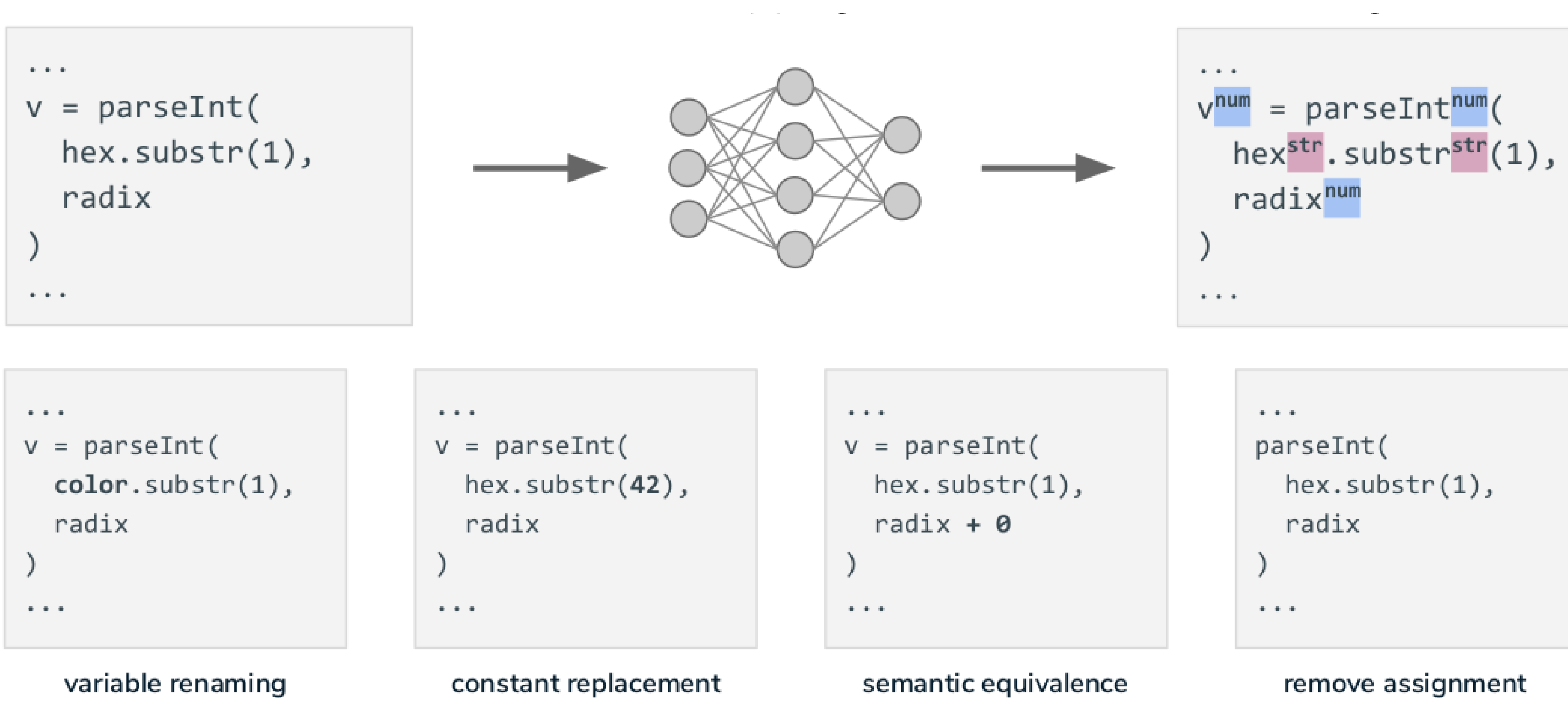
load | 93.29%

```
boolean f(Object mist) {
  for (Object elem: this.elements) {
    if (elem.equals(mist)) {
      return true;
    }
  }
  return false;
}
```

fOo | 35.77%

Improving Robustness

- **Goal: Model is correct for all label-preserving code transformations**
- **Example: Type prediction**



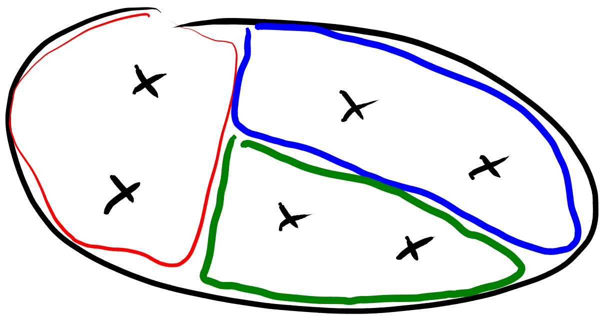
Following slides based on: “Adversarial Robustness for Code”, Bielik et al., 2020

Four Techniques

- Abstain from making a prediction
- Adversarial training
- More robust representation learning
- Train multiple specialized models

Abstaining from Making a Prediction

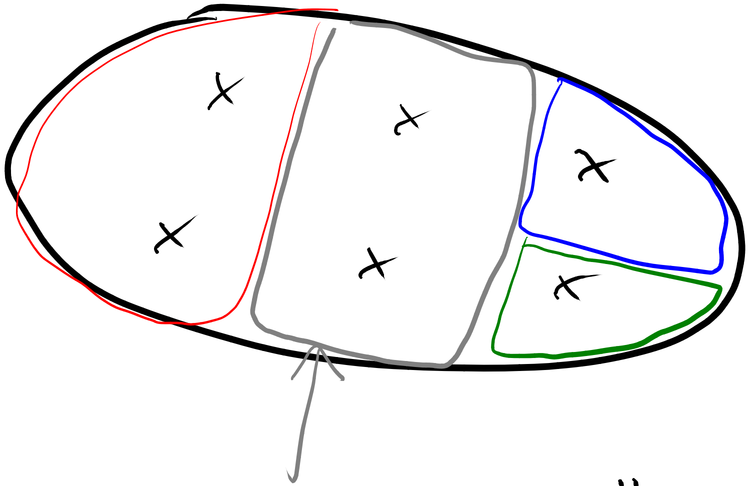
Usually:



× ... example to classify

→ Model is forced to classify each example

Instead:



New "don't know" class (with constant, small loss)

→ Simpler prediction problem

Adversarial Training

- **Label-preserving transformations**

Constants, Binary Operators, ...

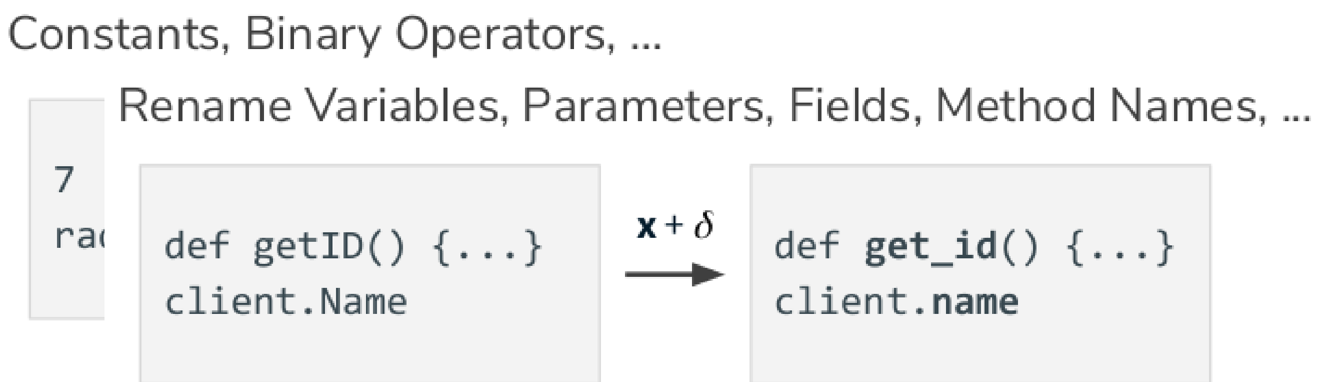
```
7
radix + offset
```

$\xrightarrow{\mathbf{x} + \delta}$

```
42
radix - offset
```

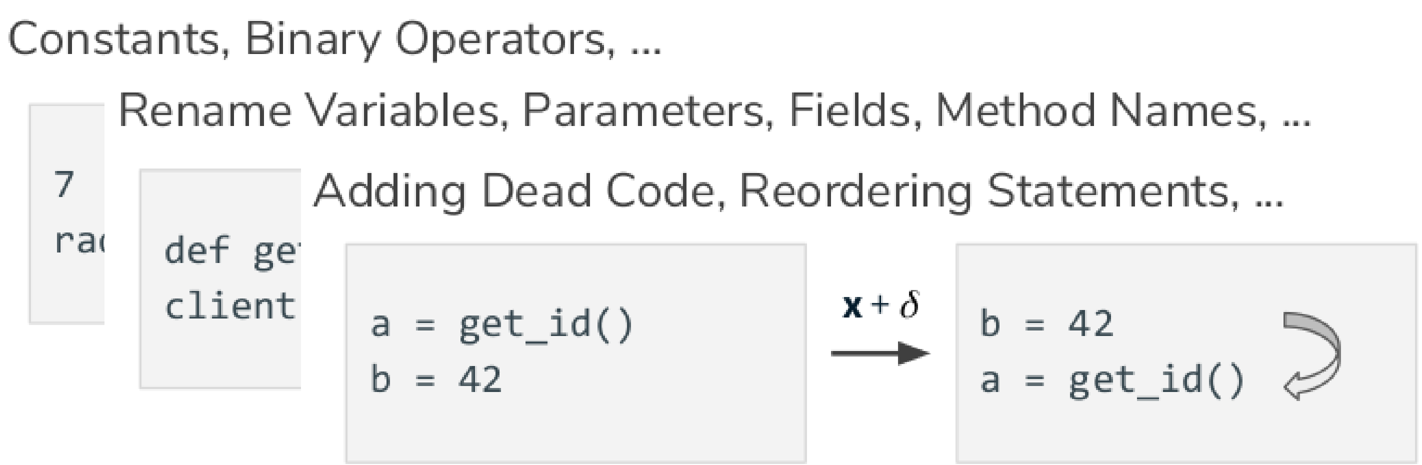
Adversarial Training

- **Label-preserving transformations**



Adversarial Training

- **Label-preserving transformations**



- **Optimization objective**:
  Minimize the maximum loss obtained by any transformation

Multiple Specialized Models

- **Train multiple models**
  - Each specializing on specific kinds of programs
- **Algorithm**
  - Train model m_i
  - Remove all data m_i is successful on
  - Train another model m_{i+1}
  - Repeat until overall accuracy high enough

Results

- **Applied to type prediction problem**
- **Three models**
  - LSTM on tokens
  - LSTM on sequentialized AST node
  - GNN
- **Large increase of robustness**
  - E.g., +29% for GNN model
- **Minor decrease of accuracy**
  - E.g., -1% for GNN model

Overview

- **Robustness**
- **Explaining specific predictions** ←
- **Explaining entire models**

Recommended papers:

- "Adversarial Examples for Models of Code", Yefet et al., 2020
- "Counterfactual Explanations for Models of Code", Cito et al., 2022
- "Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code", Paltenghi et al., 2021

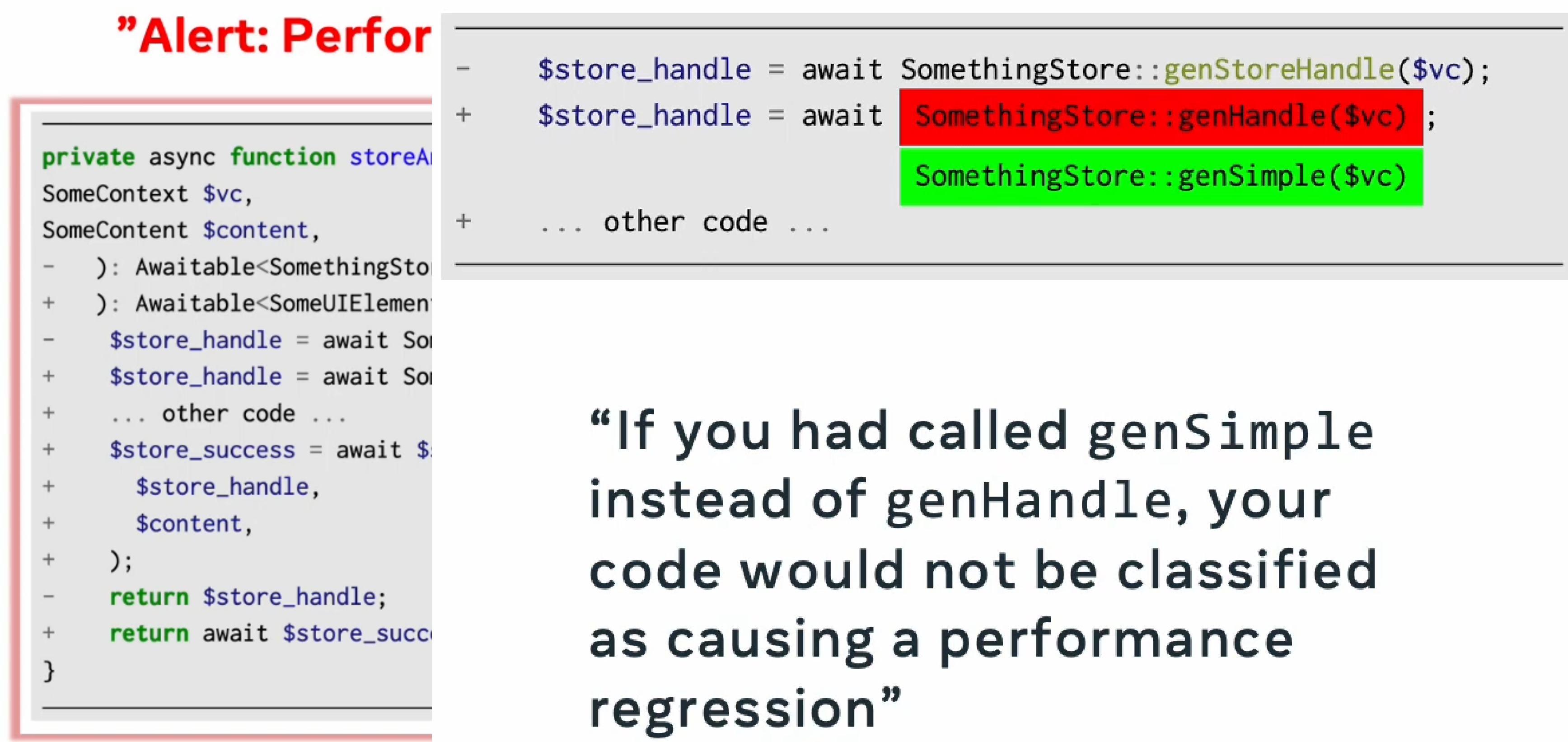
Counterfactual Explanations

"Alert: Performance regression!"

```
private async function storeAndDisplayDialog(
SomeContext $vc,
SomeContent $content,
-   ): Awaitable<SomethingStoreHandle> {
+   ): Awaitable<SomeUIElement> {
-    $store_handle = await SomethingStore::genStoreHandle($vc);
+    $store_handle = await SomethingStore::genHandle($vc);
+    ... other code ...
+    $store_success = await $store_handle->store(
+      $store_handle,
+      $content,
+    );
-    return $store_handle;
+    return await $store_success->genUIElementToRender();
}
```

Problem: Prediction alone (even if correct) may not convince developers

Counterfactual Explanations



Problem: Prediction alone (even if correct) may not convince developers

Instead: Show alternative input that changes the prediction

Goals

- **Based on feedback by software engineers at Meta**
  - Plausability: Does the counterfactual look like natural code?
  - Actionability: Does the explanation show a potential fix?
  - Consistency: Are changed applied consistently across the entire program?

Importance of Plausability

- **Counterfactual must be plausible (or natural)**
- **Otherwise:**
  - Model's prediction may be unreliable (because out-of-distribution)
  - Developers don't believe the explanation
  - Developers don't care about the explanation

Pertubation via MLM (Masked Language Model)

- Replace a token with [MASK]
- Ask a language model to **predict likely replacements for [MASK]**
- If a likely replacement **changes the prediction**: Found counterfactual
- Otherwise: Keep searching by expanding promising replacements with more tokens

Results

- **Applied to three tasks**
  - Predict performance regressions
  - Predict whether a test plan needs a screenshot
  - Predict whether a commit introduces a taint flow
- **Feedback from software engineers**
  - 83% of explanations are useful
  - Explanations help in discerning true/false positive predictions with 87% accuracy

Overview

- **Robustness**
- **Explaining specific predictions**
- **Explaining entire models** ←

Recommended papers:

- "Adversarial Examples for Models of Code", Yefet et al., 2020
- "Counterfactual Explanations for Models of Code", Cito et al., 2022
- "Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code", Paltenghi et al., 2021

Developers vs. Neural Models

Do neural models reason about code similarly to human developers?

- If yes: Good news
- If no: Should mimic developers more closely

Idea: Compare Humans & Models

Developers

vs.

Machine Learning

Neural models of code

- Same task
- Same code examples
- Measure attention and effectiveness

Task: Code Summarization

```
{
  if (!prepared(state)) {
    return state.setStatus(MovementStatus.PREPPING);
  } else if (state.getStatus() == MovementStatus.PREPPING) {
    state.setStatus(MovementStatus.WAITING);
  }
  if (state.getStatus() == MovementStatus.WAITING) {
    state.setStatus(MovementStatus.RUNNING);
  }
  return state;
}
```

Input: Method body → **Output: Method name**

`updateState`

Dataset: 250 methods from 10 Java projects *

* *A Convolutional Attention Network for Extreme Summarization of Source Code*, ICML'16

Capturing Human Attention

- **Goal: Track human attention while performing the task**
- **Approach: Unblurring-based web interface**
  - Initially, all code blurred
  - Moving mouse/cursor temporarily unblurs tokens

Model Attention

- **Convolutional sequence-to-sequence (CNN)**

  A Convolutional Attention Network for Extreme Summarization of Source Code, ICML'16
- **Transformer-based, sequence-to-sequence model**

  A Transformer-based Approach for Source Code Summarization, ACL'20
- **Both models: Regular attention and copy attention**

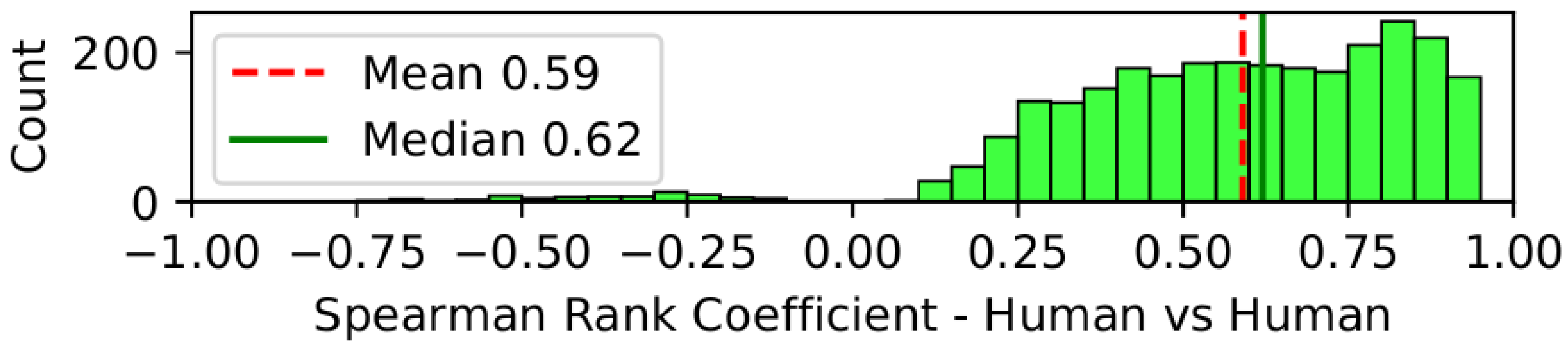
Human-Model Agreement

Do developers and models focus on the same tokens?

- Given for each code example
  - Human attention vector $\vec{h}$
  - Model attention vector $\vec{m}$
- Measure agreement between them
  - Spearman rank correlation: $\frac{cov(rg_{\vec{h}}, rg_{\vec{m}})}{\sigma_{rg_{\vec{h}}}, \sigma_{rg_{\vec{m}}}}$

Results: Agreement

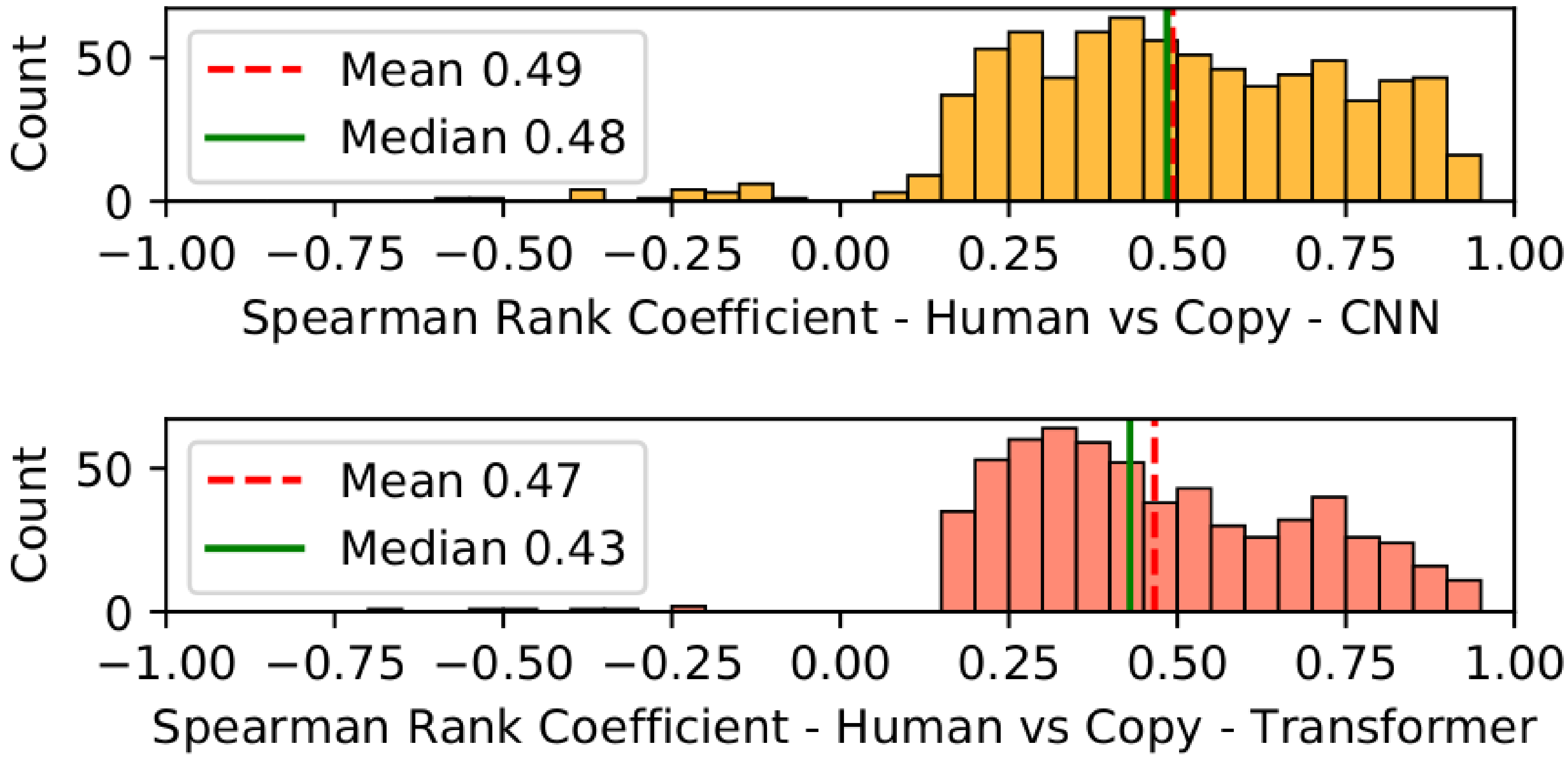
Human-human agreement:



Developers mostly agree on what code matters most

Results: Agreement

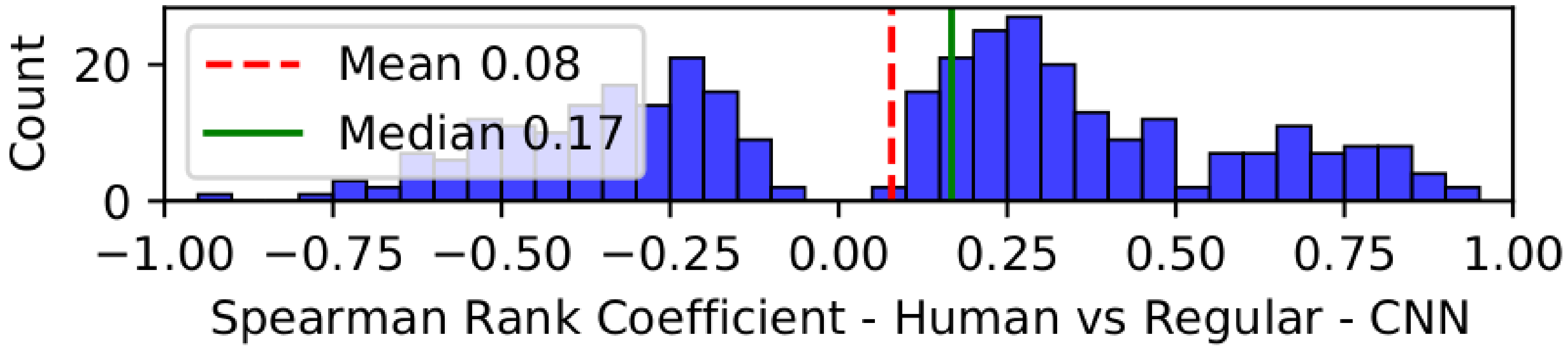
Human vs. copy attention:

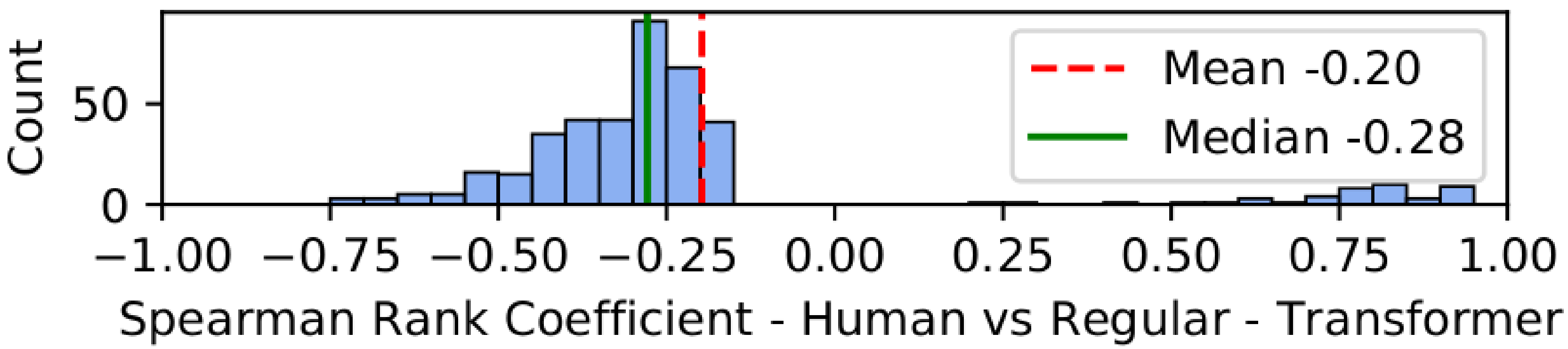


Empirical justification for copy attention

Results: Agreement

Humans vs. regular attention:





Lots of room for improvement!

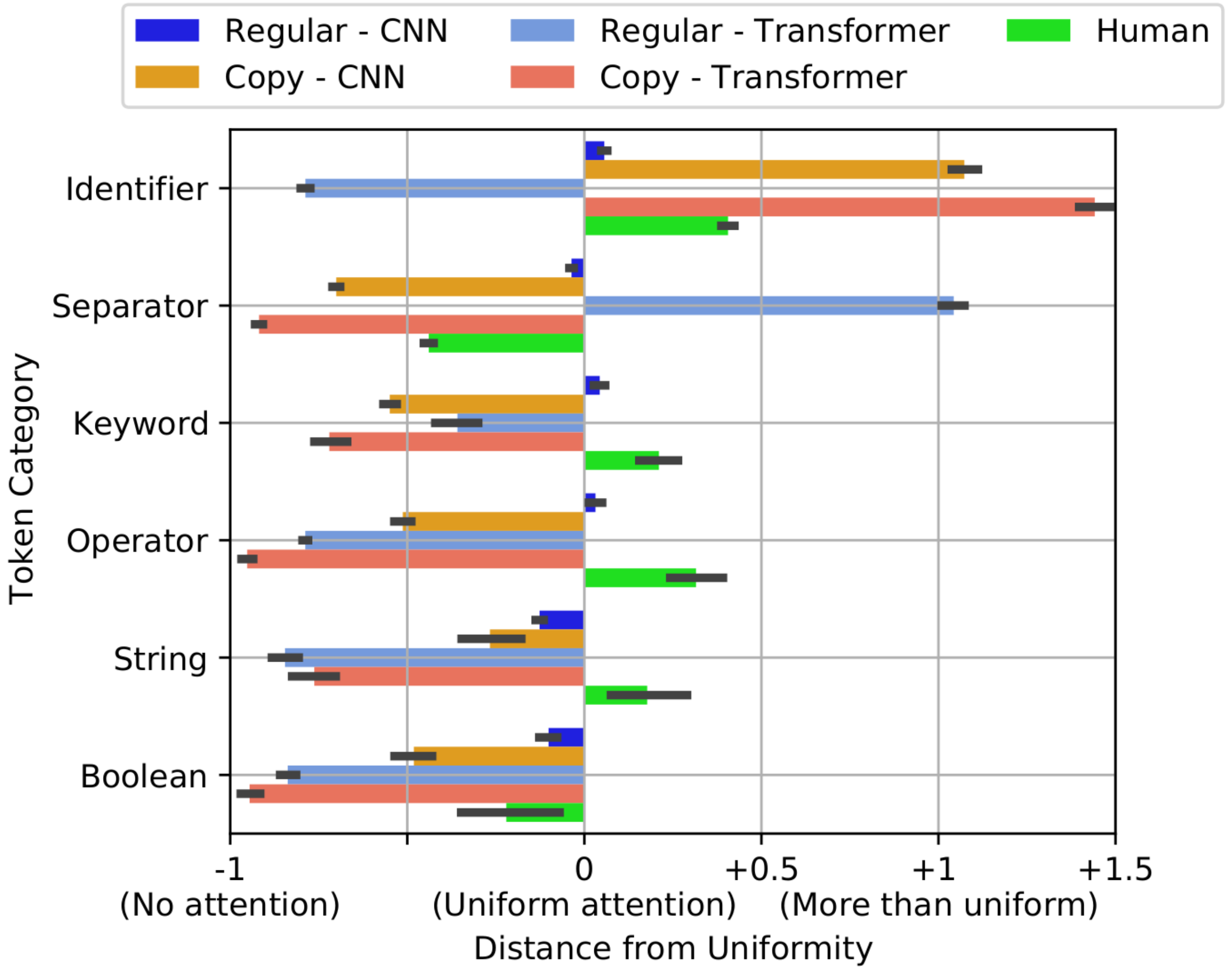
Tokens to Focus On

What kind of tokens to focus on?

- Different kinds: Identifiers, separators, etc.
- For each kind, compute distance from uniformity
  - $= 0$ means uniform attention
  - -1 means no attention at all
  - > 0 means more than uniform attention

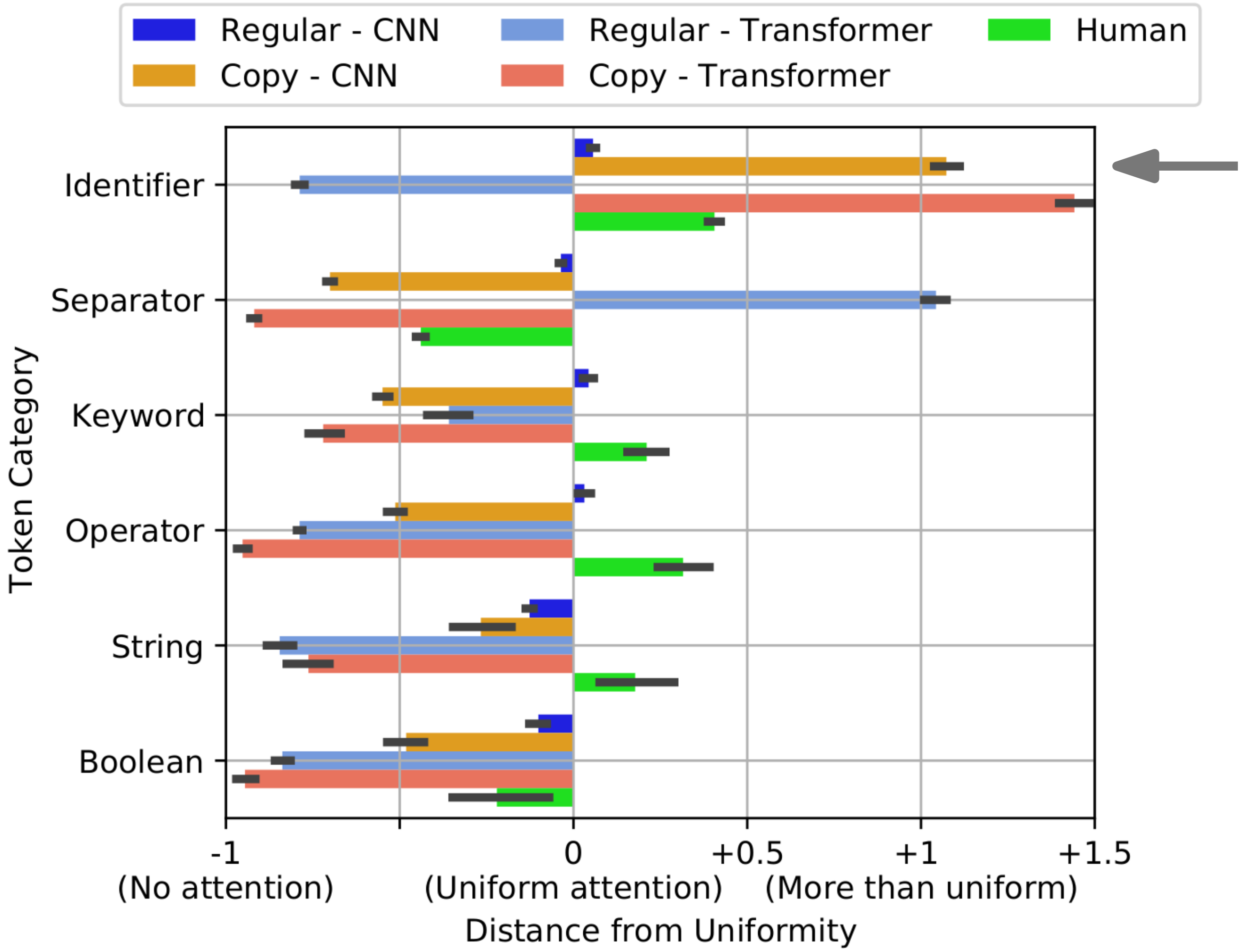
Results: Tokens to Focus on

Distance from uniformity:



Results: Tokens to Focus on

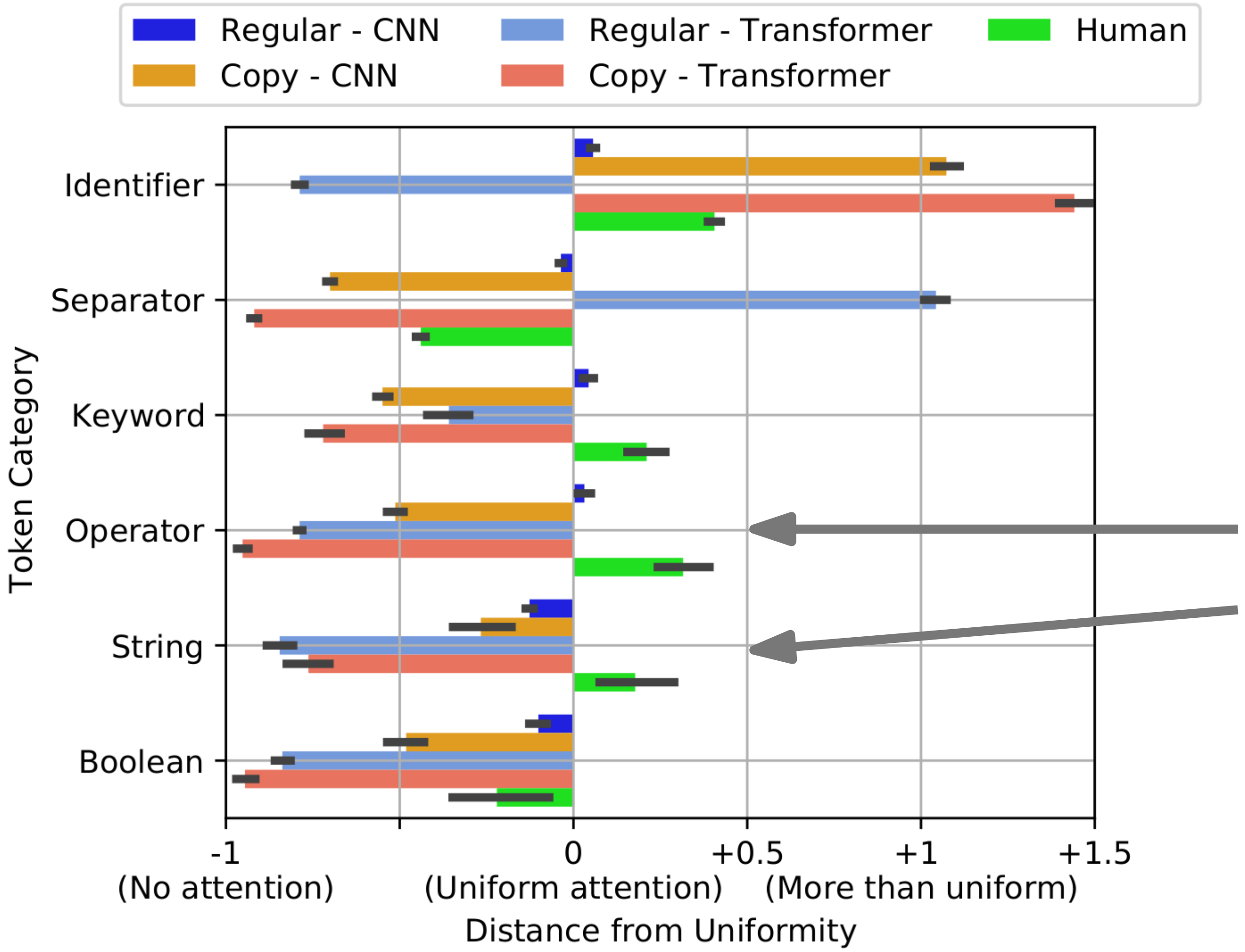
Distance from uniformity:



Identifiers are deemed important

Results: Tokens to Focus on

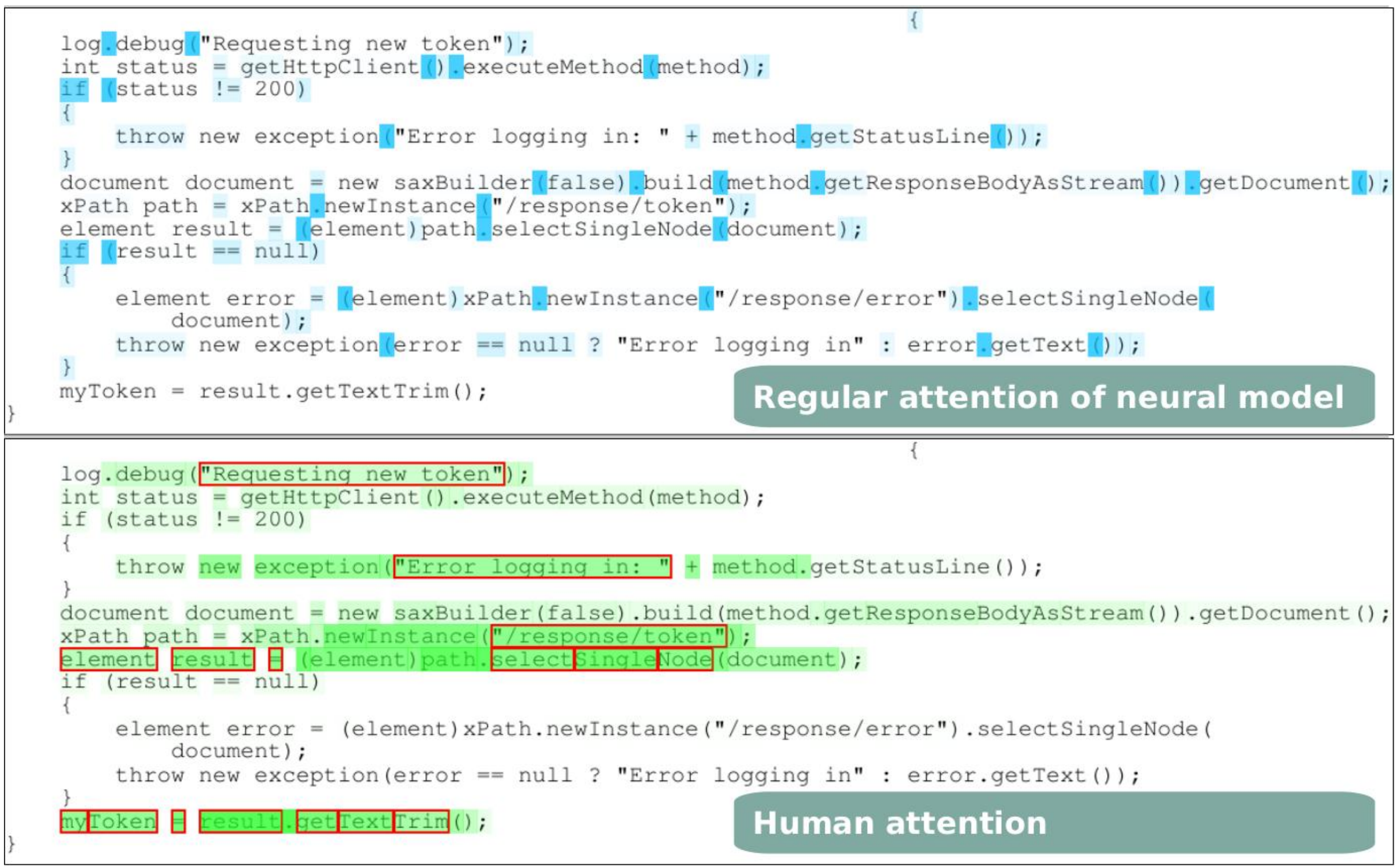
Distance from uniformity:



Models mostly ignore some kinds of tokens

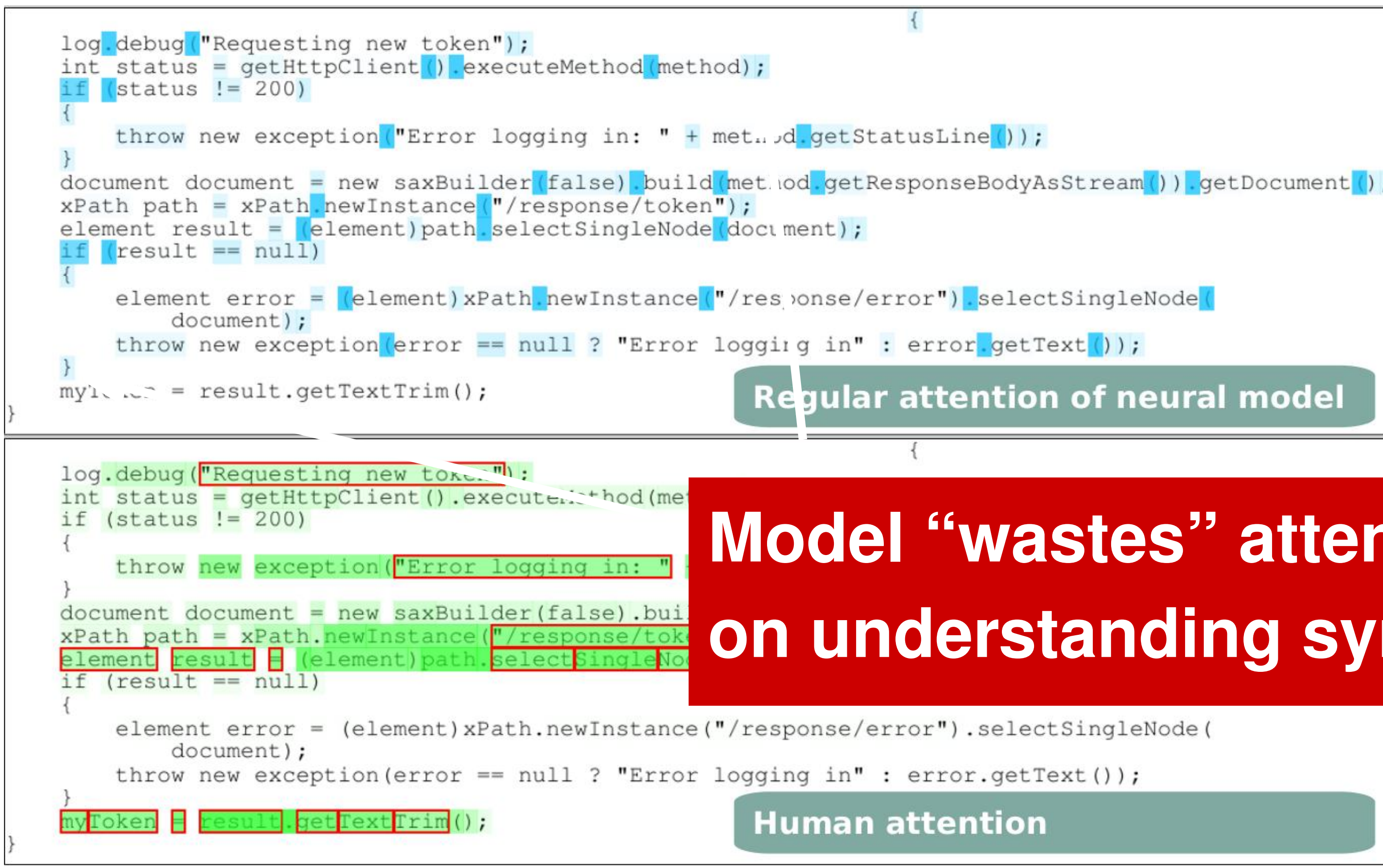
Results: Tokens to Focus on

Example from Transformer model:



Results: Tokens to Focus on

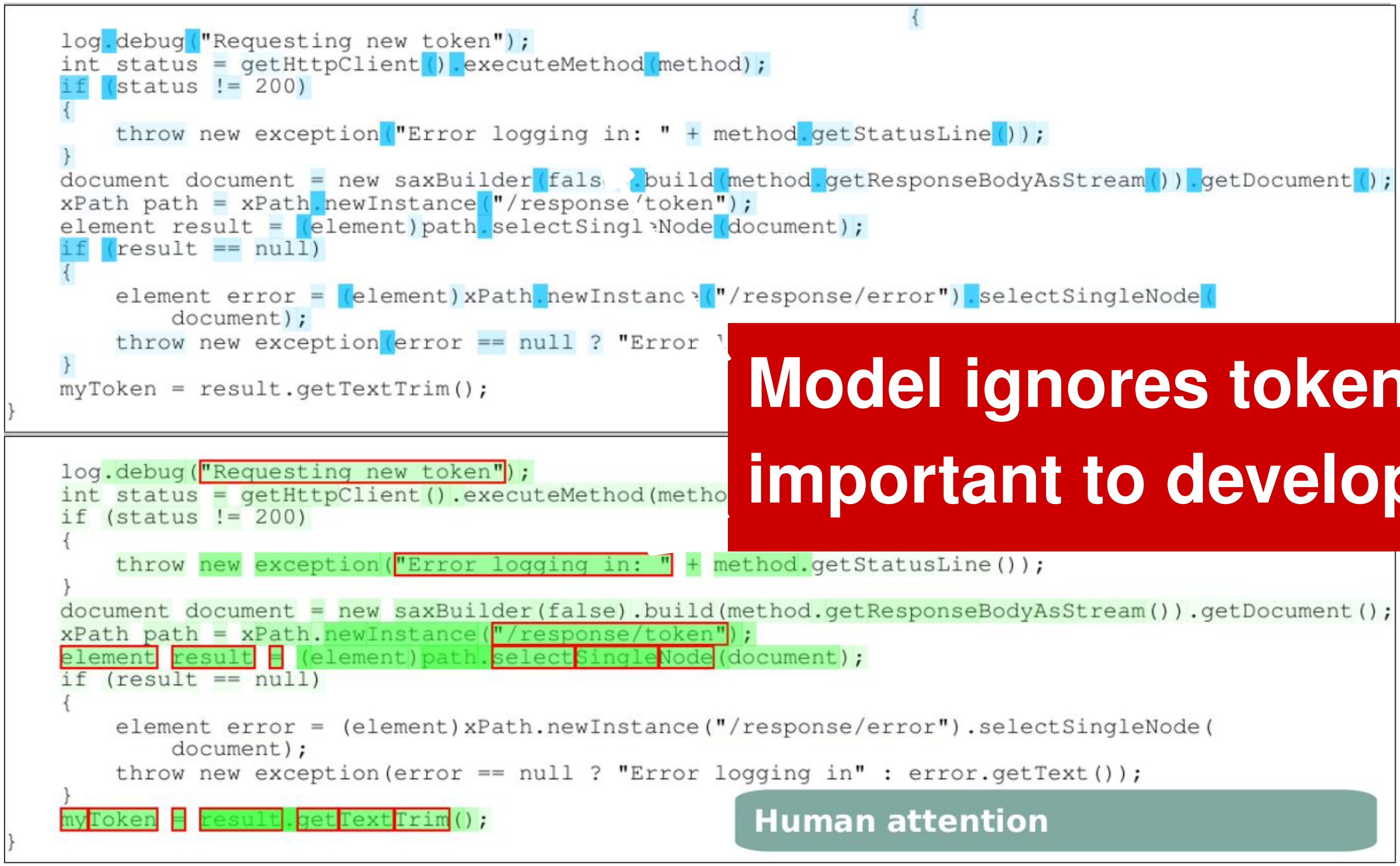
Example from Transformer model:



Model “wastes” attention on understanding syntax

Results: Tokens to Focus on

Example from Transformer model:



Model ignores tokens important to developers

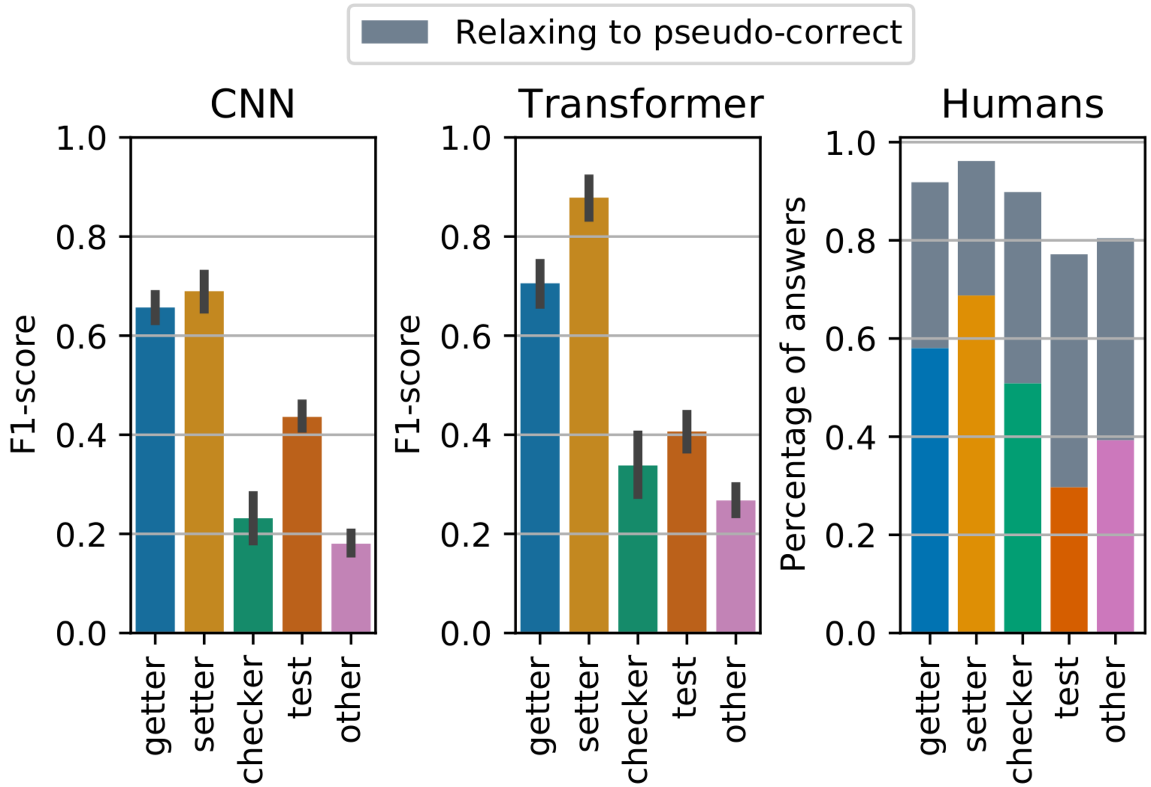
Effectiveness

Comparing developers and models w.r.t. their effectiveness at solving the task

- Strengths and weaknesses?
- Can current models compete with developers?

Results: Effectiveness

Comparing different kinds of methods:



Models underperform on non-trivial methods

Effectiveness vs. Agreement

Are models more effective when they agree more with developers?

Results: Effectiveness vs. Agreement

Human-model agreement for all vs. accurate predictions:

	Spearman rank correl.	
	All methods	Methods with F1 $\geq$ 0.5
CNN (regular)	0.08	0.24
CNN (copy)	0.49	0.55
Transformer (reg.)	-0.20	0.02
Transformer (copy)	0.47	0.55

More human-like predictions are more accurate

Implications

- **Direct human-model comparison**
  - Helps understand why models (do not) work
- **Should create models that mimic humans**
  - Use human attention during training
  - Design models that address current weaknesses
    - E.g., understanding string literals

Overview

- **Robustness**
- **Explaining specific predictions**
- **Explaining entire models** ✔

Recommended papers:

- "Adversarial Examples for Models of Code", Yefet et al., 2020
- "Counterfactual Explanations for Models of Code", Cito et al., 2022
- "Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code", Paltenghi et al., 2021