

Neural Software Analysis: Learning Developer Tools from Code



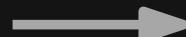
Michael Pradel

Software Lab – University of Stuttgart

**Joint work with Jibesh Patra, Georgios Gousios, Jason Liu,
and Satish Chandra**

Developers Need Tools

Key feature of humans:
Ability to develop tools



Software development tools, e.g., compilers, bug detection, code completion, documenting software

Creating Developer Tools

Traditional program analysis

- Manually crafted
- Years of work
- Precise, logical reasoning
- Heuristics to handle undecidability
- Challenged by large code bases

Creating Developer Tools

Traditional program analysis

- Manually crafted
- Years of work
- Precise, logical reasoning
- Heuristics to handle undecidability
- Challenged by large code bases



Neural software analysis

- Automatically learned within hours
- Data-driven prediction
- Learn instead of hard-code heuristics
- Use big code to our benefit

Learning Developer Tools

Insight: Lots of **data** about **software development** to **learn from**

Source code

Execution traces

Documentation →

Bug reports

etc.

Machine
Learning

Predictive
tool

Learning Developer Tools

Insight: Lots of **data** about **software development** to **learn from**

Source code

Execution traces

Documentation →

Bug reports

etc.

Machine
Learning

New code,
execution,
etc.

Predictive
tool

Information
useful for
developers

This Talk

Neural software analysis

- 1) Nalin: **Name-value inconsistencies**
- 2) TypeWriter: **Type prediction**

More details:

- *Neural Software Analysis*, CACM 2022
- *Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks*, ICSE 2022
- *TypeWriter: Neural Type Prediction with Search-based Validation*, FSE 2020

Motivation

```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

Motivation

Incorrect value:

135 . 0, should be 135



```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

Motivation

Incorrect value:

135 . 0, should be 135



```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

```
file = os.path.exists('reference.csv')
if file == False:
    print('Warning: ...')
```

Motivation

```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

Incorrect value:
135 . 0, should be 135

```
file = os.path.exists('reference.csv')
if file == False:
    print('Warning: ...')
```

Misleading name:
file vs. boolean

Motivation

```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

Incorrect value:
135 . 0, should be 135

```
file = os.path.exists('reference.csv')
if file == False:
    print('Warning: ...')
```

Commonality:
Name and value
are inconsistent

Misleading name:
file vs. boolean

Goal

Finding name-value inconsistencies

Goal

Challenge 1:
Understand the
meaning of names

Finding name-value inconsistencies

Goal

Challenge 1:
Understand the
meaning of names

Challenge 2:
Understand the
meaning of values

Finding name-value inconsistencies

Goal

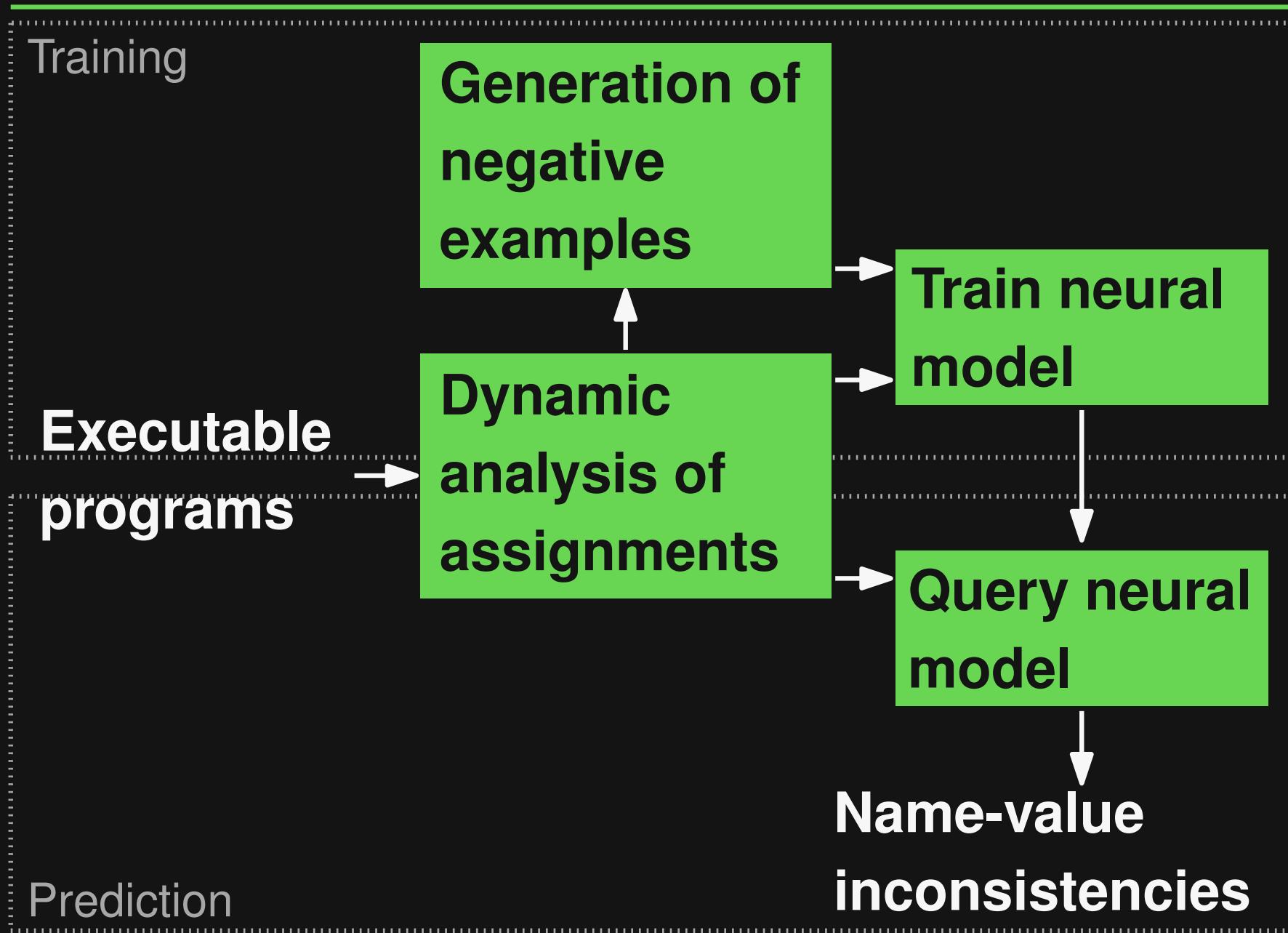
Challenge 1:
Understand the
meaning of names

Challenge 2:
Understand the
meaning of values

Finding name-value inconsistencies

Challenge 3:
Precisely pinpoint
unusual pairs

Overview of Nalin



Analyzing Assignments

Option 1: Static analysis

- 90% of assignments:

Complex expression as right-hand side

```
train_size = 0.9 * iris.data.shape[0]
```

```
file = os.path.exists('reference.csv')
```

Analyzing Assignments

Option 1: Static analysis

- 90% of assignments:
Complex expression as right-hand side

```
train_size = 0.9 * iris.data.shape[0]
```

Difficult to obtain exact value

```
file = os.path.exists('reference.csv')
```

Analyzing Assignments

Option 2: Dynamic analysis

- Source-to-source instrumentation
- Extract for each assignment
 - Name of left-hand side
 - String representation of value
 - Type of value
 - Length of value
 - Shape of value

Analyzing Assignments

Example:

Name	Value	Type	Length	Shape
age	23	int	null	null
probability	0.83	float	null	null
Xs_train	[[0.5 2.3]\n [..	ndarray	600	(600,2)
name	2.5	float	null	null
file_name	'example.txt'	str	11	null

Generation of Negative Examples

Naive approach: Randomly combine names and values

Positive examples

num=23

age=3



Negative examples

num=3

age=23

Generation of Negative Examples

Naive approach: Randomly combine names and values

Positive examples

num=23

age=3

Negative examples

→ num=3

age=23

Legitimate “negative” examples:

Noisy training data → False positives

Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is

- unusual for this name
- common across all observed values

2) Pick a random value of this type

Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is

- unusual for this name
- common across all observed values

2) Pick a random value of this type

Positive example

`years=[2011, 2012]`

Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is

- unusual for this name
- common across all observed values

2) Pick a random value of this type

Positive example

`years=[2011, 2012]`



Generation of Negative Examples

Type-guided generation:

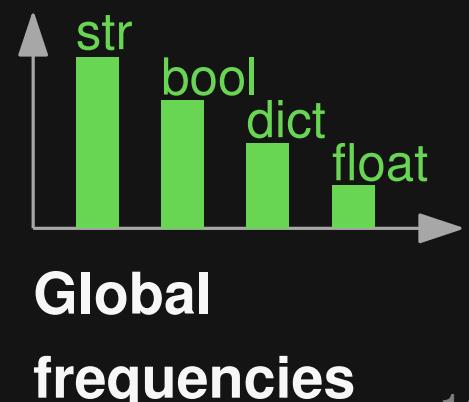
1) Given a name, select a type that is

- unusual for this name
- common across all observed values

2) Pick a random value of this type

Positive example

`years=[2011, 2012]`



Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is

- unusual for this name
- common across all observed values

2) Pick a random value of this type

Positive example

`years=[2011, 2012]`



Negative example

`years=False`

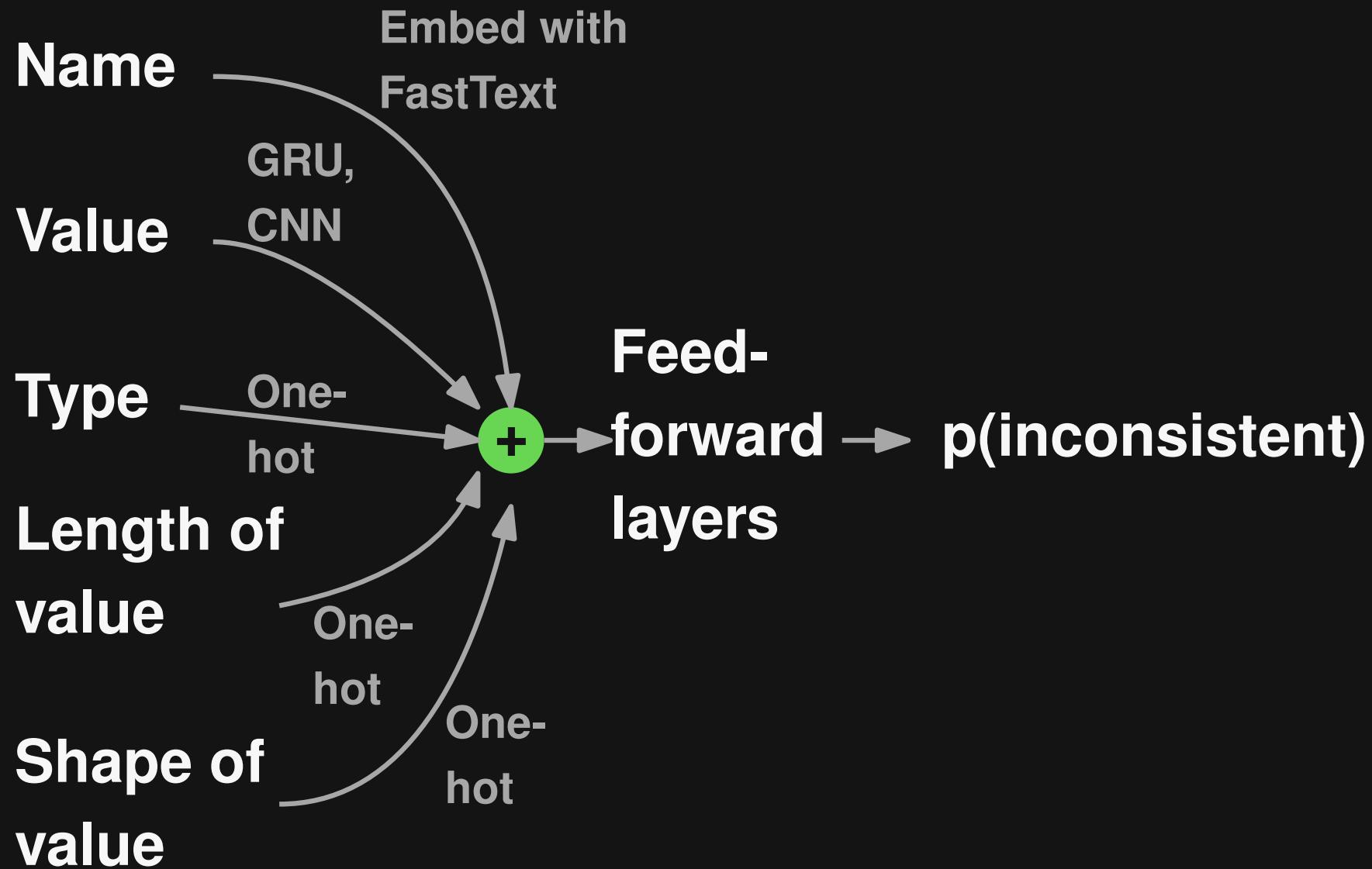


Types of
name years



Global
frequencies

Neural Classification Model



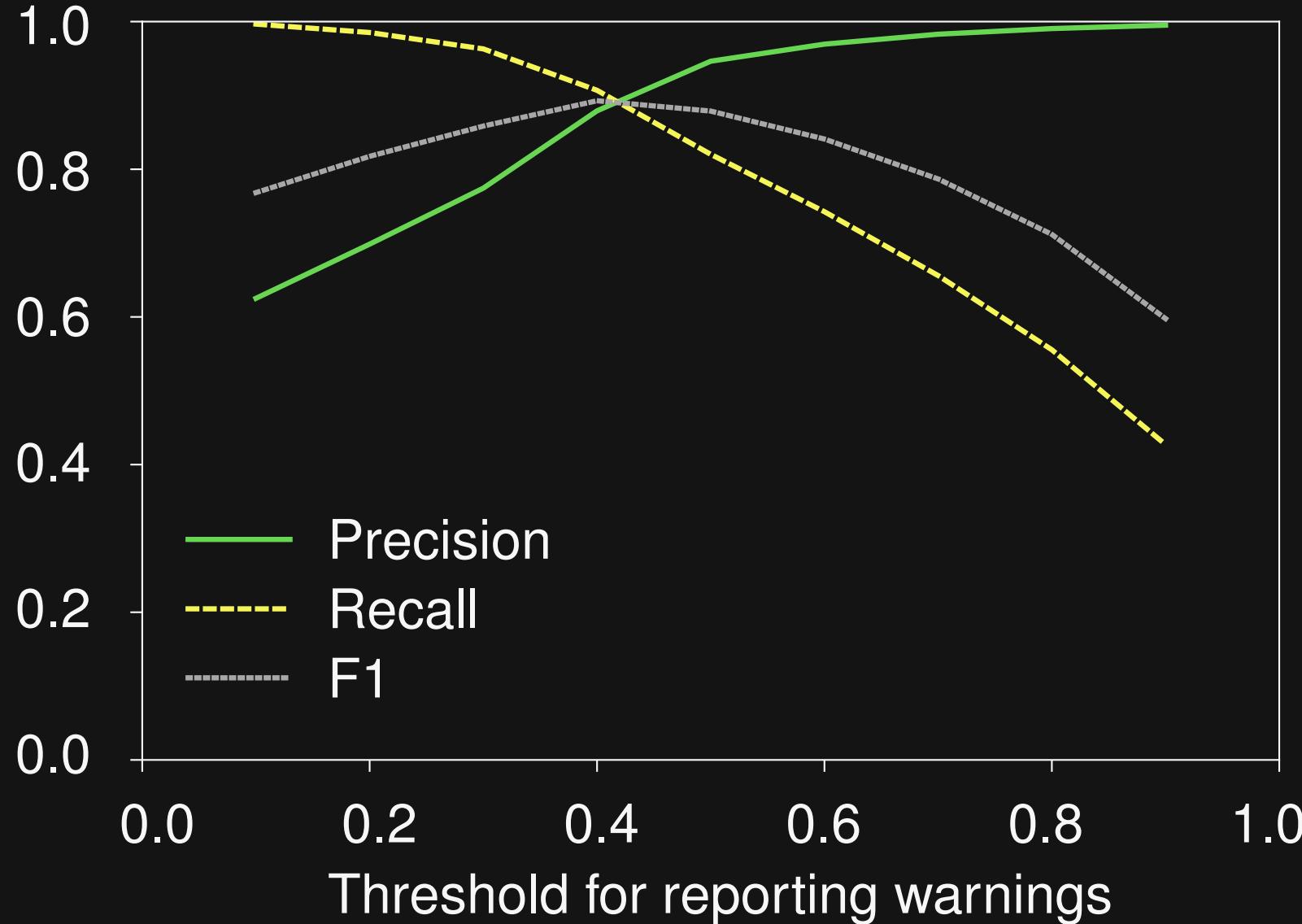
Two linear layers, 50% dropout, Adam optimizer, batch size=128

Evaluation: Setup

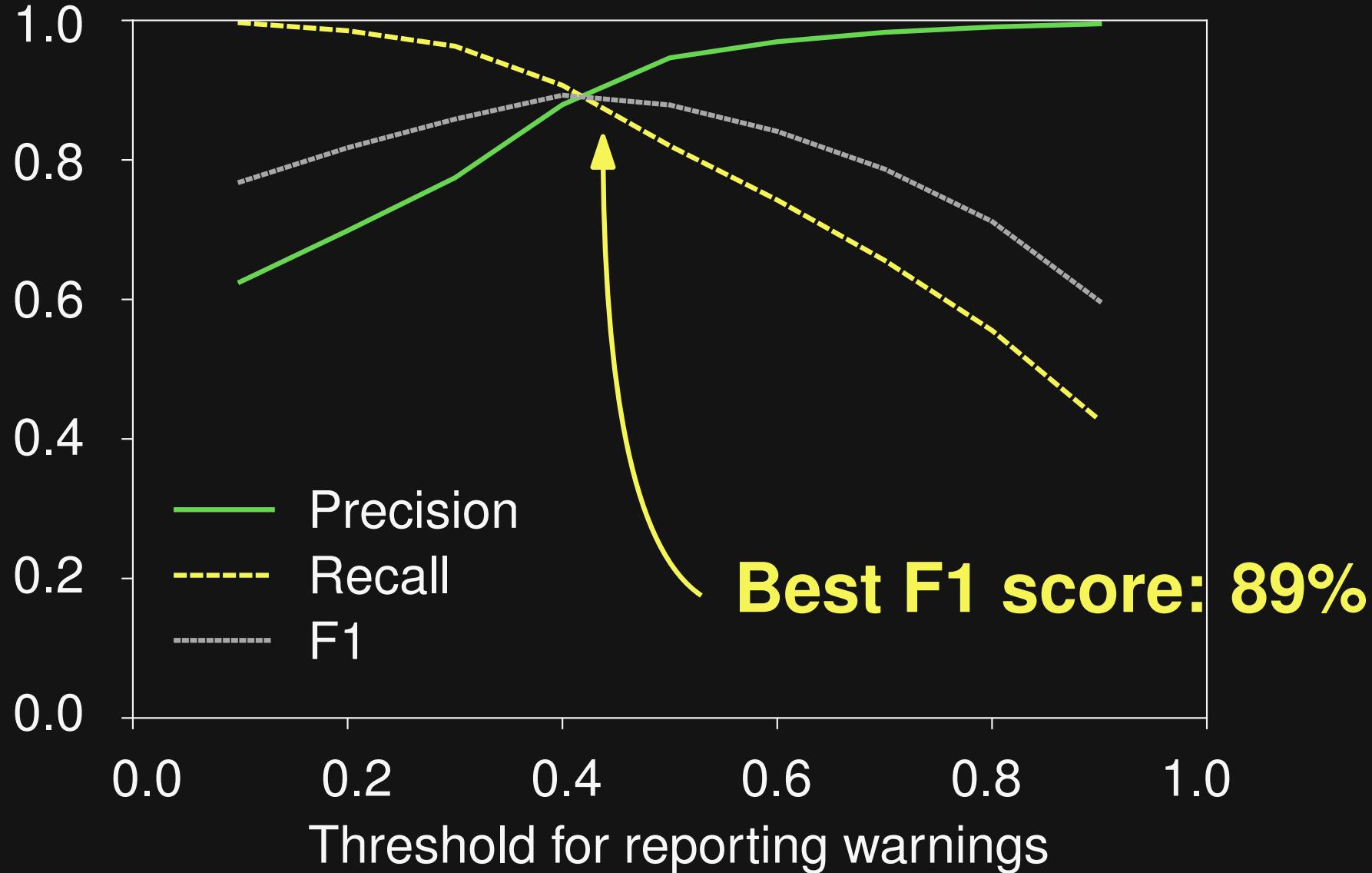
**Jupyter notebooks:
Closed, executable programs**

- Initially, 1M programs
- 106k analyzed programs (7.2MLoC)
- 947k name-value pairs

Effectiveness of Classifier



Effectiveness of Classifier



Study with Developers

Is a name-value pair **easy to understand?**

40 name-value pairs, 11 participants, 5-point Likert scale,
56% inter-rate agreement

Study with Developers

Is a name-value pair **easy to understand?**

Developer assessment	Nalin's prediction	
	Consistent	Inconsistent
Easy to understand	15	4
Hard to understand	5	16

40 name-value pairs, 11 participants, 5-point Likert scale,
56% inter-rate agreement

Study with Developers

Is a name-value pair **easy to understand?**

Developer assessment	Nalin's prediction	
	Consistent	Inconsistent
Easy to understand	15	4
Hard to understand	5	16

precision=80%, recall=76%

40 name-value pairs, 11 participants, 5-point Likert scale,
56% inter-rate agreement

Kinds of Inconsistencies

30 inspected warnings



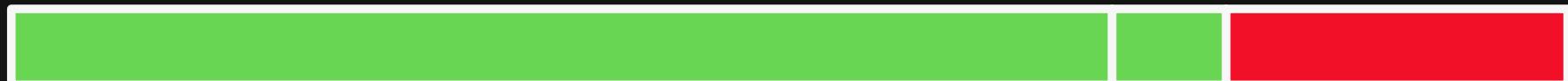
21 misleading
names

2 incorrect
values

7 false
positives

Kinds of Inconsistencies

30 inspected warnings



21 misleading
names

2 incorrect
values

7 false
positives



`name = 'Philip K. Dick'`

...

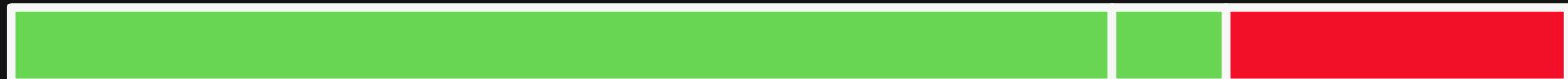
`name = 2.5`



Unusual combination

Kinds of Inconsistencies

30 inspected warnings



21 misleading
names

2 incorrect
values

7 false
positives



```
prob = get_betraying_probability(information)
if prob > 1/2:
    return D
```

Value: "Corporate"



Kinds of Inconsistencies

30 inspected warnings



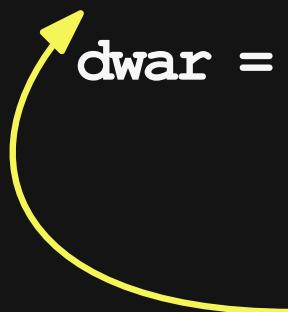
21 misleading
names

2 incorrect
values

7 false
positives



```
dwarf = '/Users/iayork/Downloads/dwarf_2013_2015.txt'  
dwarf = pd.read_csv(dwarf, sep=' ', header=None)
```



**Model doesn't understand the
abbreviation ("F" means "file")**

Comparison with Prior Tools

Approach	Warnings	
	Total	Shared with Nalin
pyre	54	1/30
flake8	1,247	0/30
DeepBugs	151	0/30

Each tool applied to 30 files with manually inspected Nalin warnings

Comparison with Prior Tools

Approach	Warnings	
	Total	Shared with Nalin
pyre	54	1/30
flake8	1,247	0/30
DeepBugs	151	0/30

Traditional static checkers

Each tool applied to 30 files with manually inspected Nalin warnings

Comparison with Prior Tools

Approach	Warnings	
	Total	Shared with Nalin
pyre	54	1/30
flake8	1,247	0/30
DeepBugs	151	0/30



Learning-based detector of name-related bugs [OOPSLA'18]

Each tool applied to 30 files with manually inspected Nalin warnings

Comparison with Prior Tools

Approach	Warnings	
	Total	Shared with Nalin
pyre	54	1/30
flake8	1,247	0/30
DeepBugs	151	0/30

Variable name holds
string and then float

Each tool applied to 30 files with manually inspected Nalin warnings

Summary: Nalin

- Automatic detection of **name-value inconsistencies**
- Learning-based bug detection on **runtime behavior**
- Type-guided generation of **negative examples**

This Talk

Neural software analysis

- 1) Nalin: Name-value inconsistencies
- 2) TypeWriter: Type prediction

More details:

- *Neural Software Analysis*, CACM 2022
- *Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks*, ICSE 2022
- *TypeWriter: Neural Type Prediction with Search-based Validation*, FSE 2020

Types in Dynamic PLs

- **Dynamically typed languages:**
Extremely popular
- **Lack of type annotations:**
 - Type errors
 - Hard-to-understand APIs
 - Poor IDE support
- **Gradual types to the rescue**

Types in Dynamic PLs

- **Dynamically typed languages:**
Extremely popular
- **Lack of type annotations:**
 - Type errors
 - Hard-to-understand APIs
 - Poor IDE support
- **Gradual types to the rescue**

But: Annotating types is painful

How to Add Type Annotations?

- **Option 1: Static type inference**

- Guarantees type correctness, but very limited

- **Option 2: Dynamic type inference**

- Depends on inputs and misses types

- **Option 3: Probabilistic type prediction**

- Models learned from existing type annotations

Probabilistic Type Prediction

Neural model to predict types



Prior models:

- *Deep Learning Type Inference*, FSE'18
- *NL2Type: Inferring JavaScript Function Types from Natural Language Information*, ICSE'19

Challenges

■ Imprecision

- Some predictions are wrong
- Developers must decide which suggestions to follow

■ Combinatorial explosion

- For each missing type: One or more suggestions
- Exploring all combinations:
Practically impossible

Example

```
def find_match(color):
    """
Args:
    color (str): color to match on and return
"""
candidates = get_colors()
for candidate in candidates:
    if color == candidate:
        return color
return None

def get_colors():
    return ["red", "blue", "green"]
```

Example

```
def find_match(color):  
    """
```

Args:

 color (str): color to match on and return

"""

```
candidates = get_colors()
```

```
for candidate in candidates:
```

```
    if color == candidate:
```

```
        return color
```

```
return None
```

Predictions:

1) int

2) str

3) bool

Predictions:

1) str

2) Optional[str]

3) None

```
def get_colors():
```

```
    return ["red", "blue", "green"]
```

Predictions:

1) List[str]

2) List[Any]

3) str

Example

```
def find_match(color):
```

"""

Args:

color (s) Type errors

"""

```
candidates = get_colors()  
for candidate in candidates:  
    if color == candidate:  
        return color  
return None
```

Top-most predictions:

Predictions:

1) int

2) str

3) bool

and return

Predictions:

1) str

2) Optional[str]

3) None

```
def get_colors():
```

```
    return ["red", "blue", "green"]
```

Predictions:

1) List[str]

2) List[Any]

3) str

Example

```
def find_match(color):  
    """  
    Args:  
        color (str)  
    """  
  
    candidates = get_colors()  
    for candidate in candidates:  
        if color == candidate:  
            return color  
    return None
```

```
def get_colors():  
    return ["red", "blue", "green"]
```

Correct predictions

Predictions:

1) int

2) str

3) bool

and return

Predictions:

1) str

2) Optional[str]

3) None

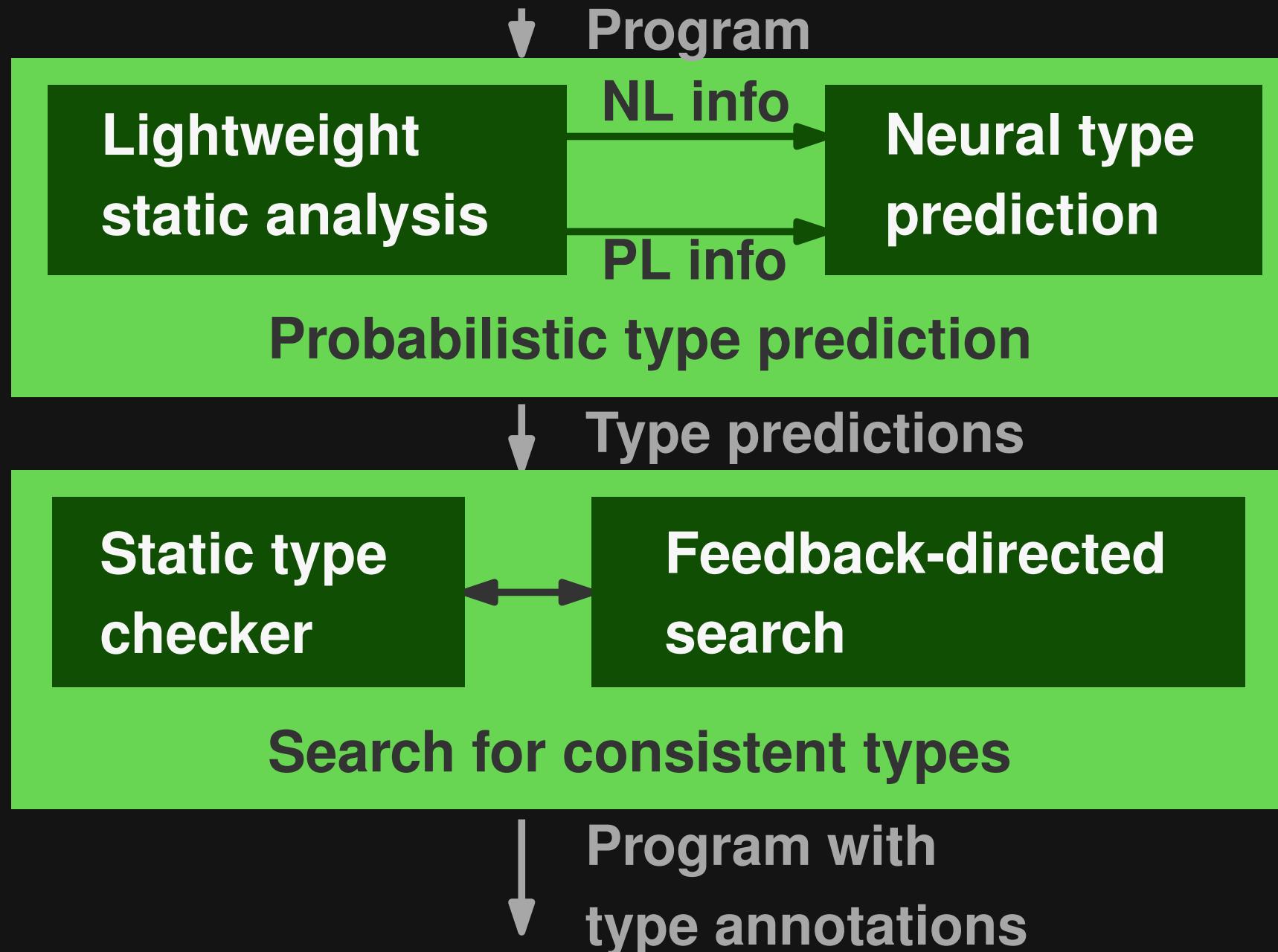
Predictions:

1) List[str]

2) List[Any]

3) str

Overview of TypeWriter



Extracting NL and PL Info

■ NL information

- Names of functions and arguments
- Function-level comments

■ PL information

- Occurrences of the to-be-typed code element
- Types made available via imports

Extracting NL Information

```
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```

Extracting NL Information

```
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """

    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```

Identifiers associated
with the to-be-typed
program element

Args:



Extracting NL Information

```
def find_match(color):
    """
Args:
    color (str) : color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```

Function-level
comments



Extracting PL Information

```
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```

Extracting PL Information

```
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None
```

Tokens around
occurrences of the
to-be-typed code element

```
def get_colors():
    return ["red", "blue", "green"]
```



Extracting PL Information

```
def find_match(color):
```

```
    """
```

Args:

 color (str) : color to match on and return

```
    """
```

```
    candidates = get_colors()
```

```
    for candidate in candidates:
```

```
        if color == candidate:
```

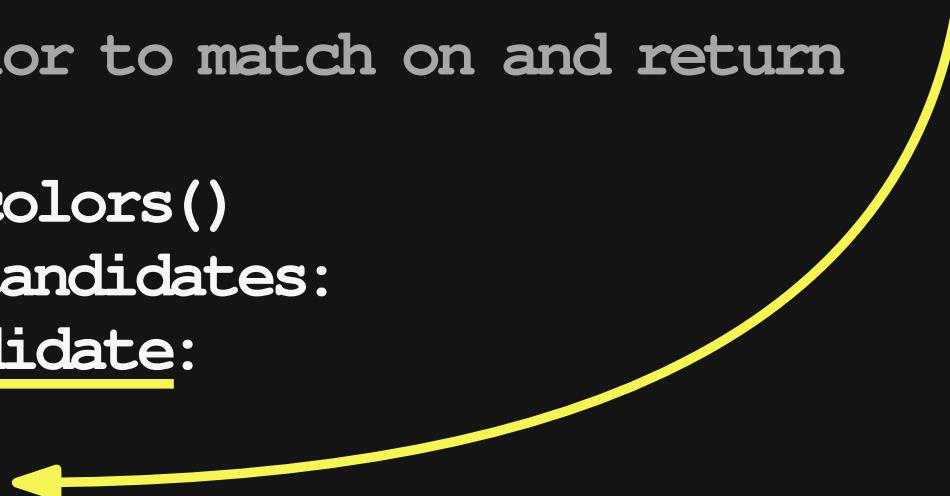
```
            return color
```

```
    return None
```

Tokens around
occurrences of the
to-be-typed code element

```
def get_colors():
```

```
    return ["red", "blue", "green"]
```



Extracting PL Information

```
from ab import de  
import x.y.z
```



Types made available via imports

```
def find_match(color):  
    """
```

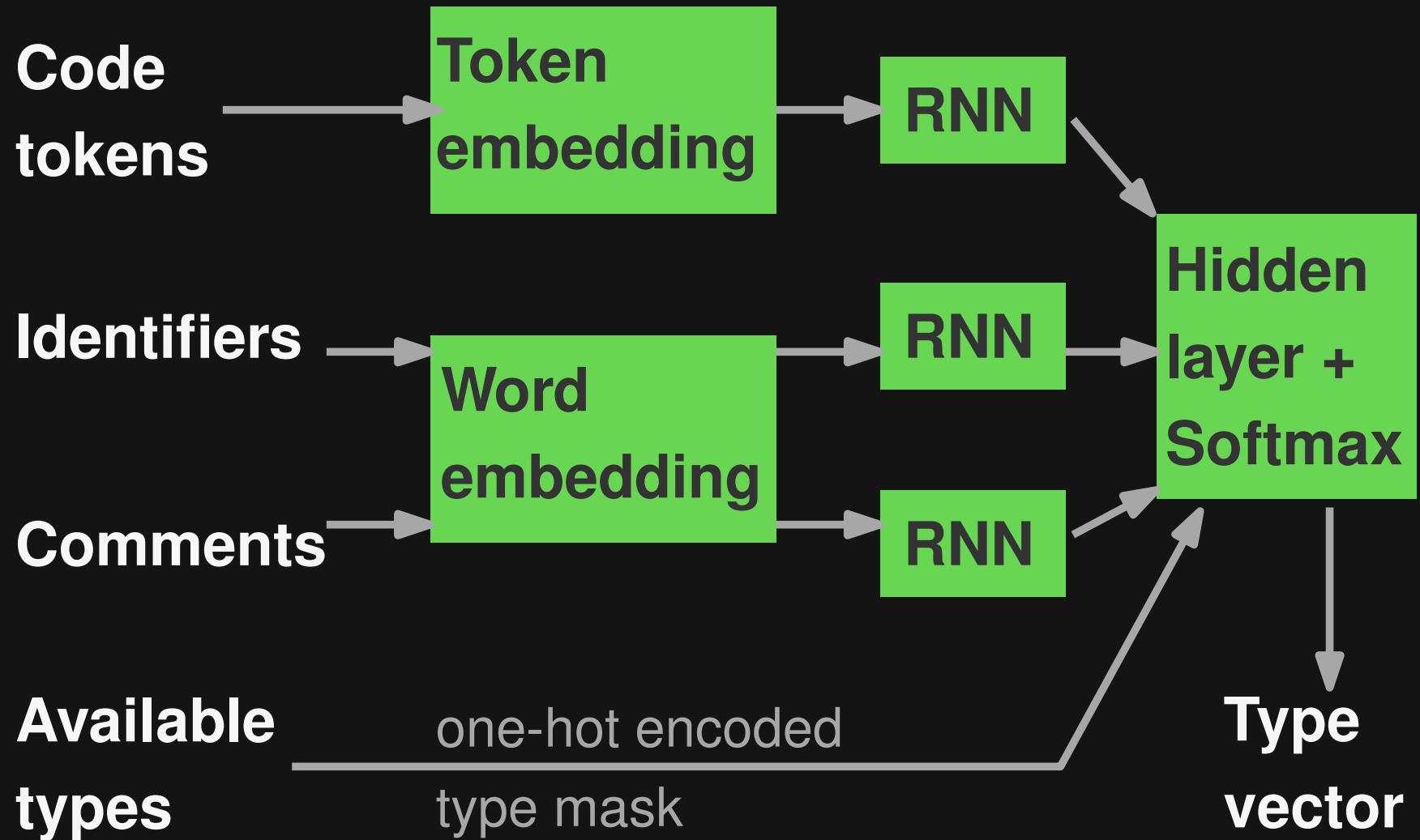
Args:

 color (str) : color to match on and return
 """

```
candidates = get_colors()  
for candidate in candidates:  
    if color == candidate:  
        return color  
return None
```

```
def get_colors():  
    return ["red", "blue", "green"]
```

Neural Type Prediction Model



Searching for Consistent Types

- **Top-k predictions for each missing type**
 - Filter predictions using gradual type checker
 - E.g., pyre and mypy for Python, flow for JavaScript
- **Combinatorial search problem**
 - For type slots S and k predictions per slot:
 $(k + 1)^{|S|}$ possible type assignments

Searching for Consistent Types

- **Top-k predictions for each missing type**

- Filter predictions using gradual type checker
 - E.g., pyre and mypy for Python, flow for JavaScript

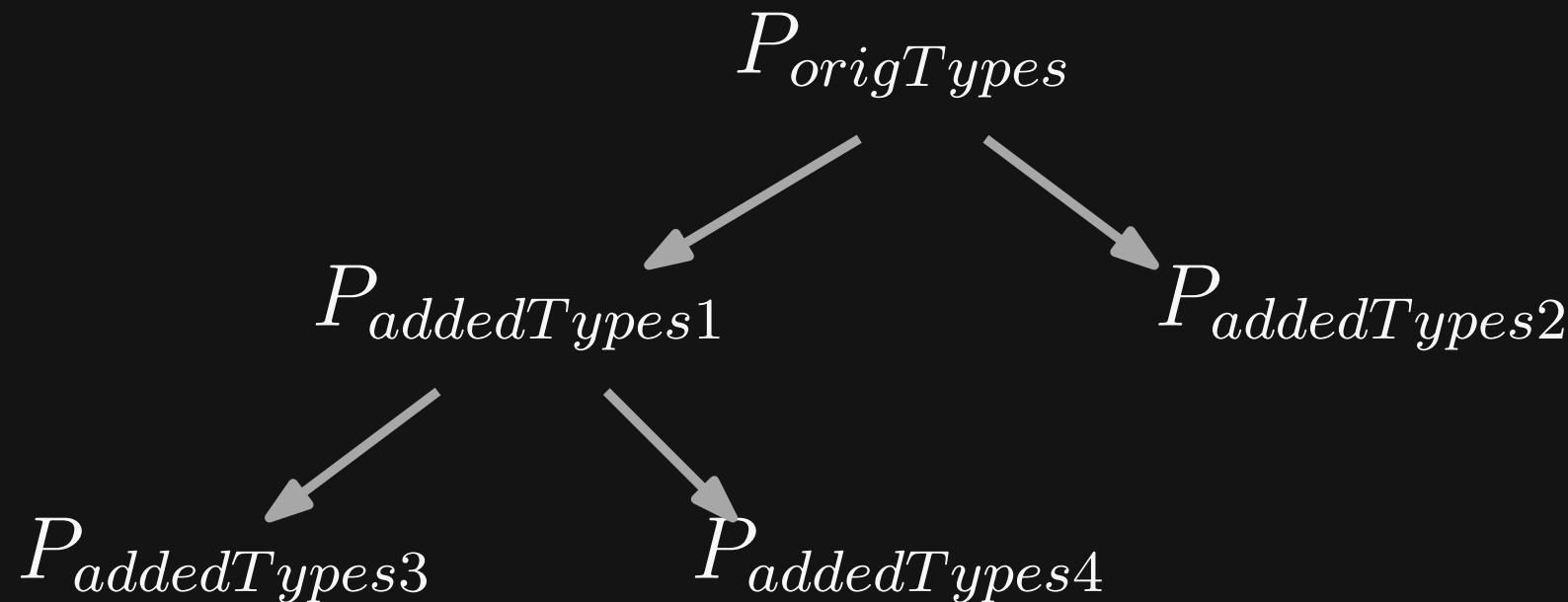
- **Combinatorial search problem**

- For type slots S and k predictions per slot:
 $(k + 1)^{|S|}$ possible type assignments

Too large to explore exhaustively!

Exploring the Search Space

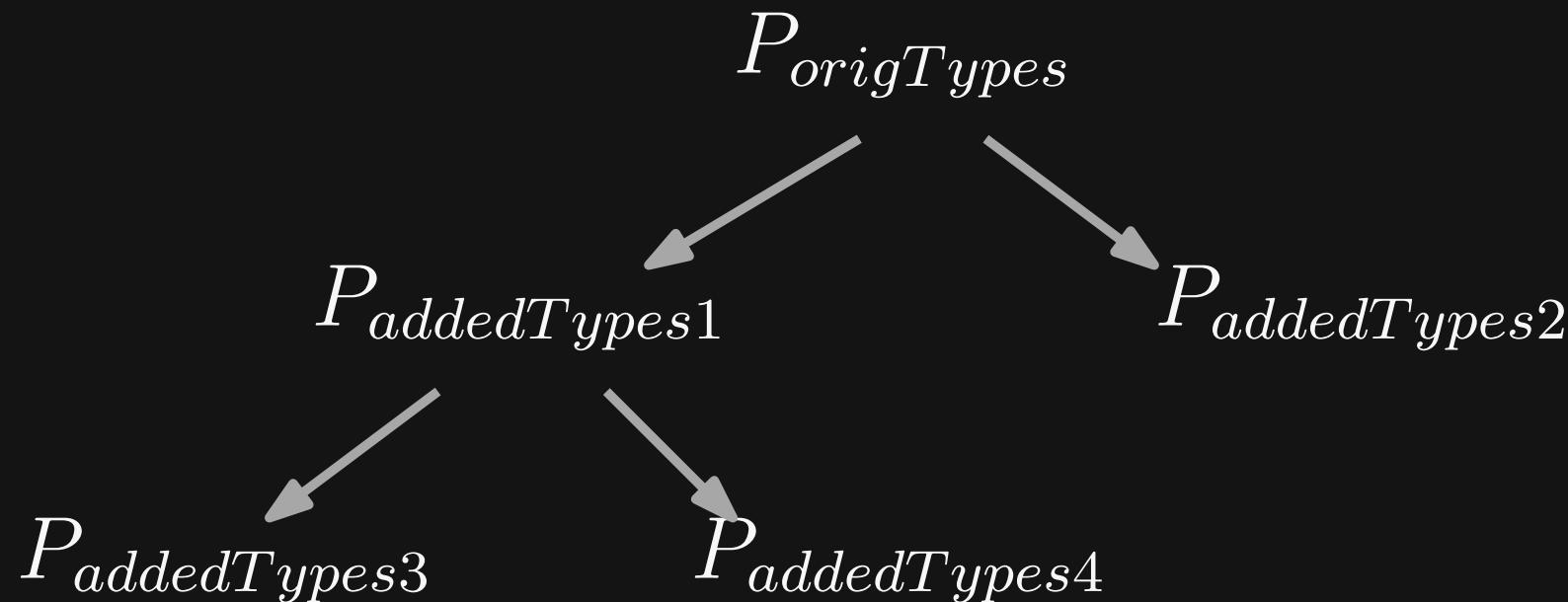
Tree of variants of program P



→ ... add, remove, or replace types

Exploring the Search Space

Tree of variants of program P



Which variants to explore first?

→ ... add, remove, or replace types

Feedback Function

- Goal: **Minimize missing types without introducing type errors**
- **Feedback score (lower is better):**

$$v \cdot n_{missing} + w \cdot n_{errors}$$

Feedback Function

- Goal: **Minimize missing types without introducing type errors**
- **Feedback score (lower is better):**

$$v \cdot n_{missing} + w \cdot n_{errors}$$



Default: $v = 1, w = 2,$
i.e., higher weight for errors

Exploring the Search Space

- **Optimistic:** Add top-most predicted type everywhere and then update types

- **Greedy or non-greedy**

↓
If score decreases,
keep the type

→ Backtrack to avoid
local minima

Example

```
def find_match(color):
    """
Args:
    color (str): color to match on and return
"""
candidates = get_colors()
for candidate in candidates:
    if color == candidate:
        return color
return None

def get_colors():
    return ["red", "blue", "green"]
```

Example

```
def find_match(color):  
    """
```

Args:

 color (str): color to match on and return

"""

```
    candidates = get_colors()  
    for candidate in candidates:  
        if color == candidate:  
            return color  
    return None
```

Predictions:

1) int

2) str

3) bool

Predictions:

1) str

2) Optional[str]

3) None

```
def get_colors():  
    return ["red", "blue", "green"]
```

Predictions:

1) List[str]

2) List[Any]

3) str

Example

```
def find_match(color):  
    """
```

Args:

 color (str): color to match on and return

"""

```
candidates = get_colors()
```

```
for candidate in candidates:
```

```
    if color == candidate:
```

```
        return color
```

```
return None
```

Predictions:

1) int

2) str

3) bool

Predictions:

1) str

2) Optional[str]

3) None

```
def get_colors():
```

```
    return ["red", "blue", "green"]
```

Predictions:

1) List[str]

2) List[Any]

3) str

Example

```
def find_match(color):
```

```
    """
```

Args:

 color (str) : color to match on and return

```
    """
```

```
    candidates = get_colors()
```

```
    for candidate in candidates:
```

```
        if color == candidate:
```

```
            return color
```

```
    return None
```

```
def get_colors():
```

```
    return ["red", "blue", "green"]
```

Predictions:

1) int

2) str

3) bool

Predictions:

1) str

2) Optional[str]

3) None

Predictions:

1) List[str]

2) List[Any]

3) str

Example

```
def find_match(color):  
    """
```

Args:

 color (str): color to match on and return

"""

```
    candidates = get_colors()  
    for candidate in candidates:
```

 if color == candidate:

 return color

```
    return None
```

```
def get_colors():
```

```
    return ["red", "blue", "green"]
```

Predictions:

1) int

2) str

3) bool

Predictions:

1) str

2) Optional[str]

3) None



Predictions:

1) List[str]

2) List[Any]

3) str

Evaluation: Setup

■ Code corpora

- Facebook's Python code
- 5.8 millions lines of open-source code

■ Types

- Millions of argument and return types
- 6-12% already annotated
- Trivial types (e.g., type of `self`) ignored

Effectiveness of Neural Model

Approach	Top-1		
	Prec	Rec	F1
TypeWriter	65%	59%	62%

Effectiveness of Neural Model

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
TypeWriter	65%	59%	62%	80%	71%	75%	85%	75%	80%

Effectiveness of Neural Model

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
TypeWriter	65%	59%	62%	80%	71%	75%	85%	75%	80%
NL2Type	59%	55%	57%	73%	67%	70%	79%	71%	75%
Frequencies	12%	20%	15%	19%	35%	25%	22%	39%	28%

Effectiveness of Search

Strategy	Top- k	Annotations	
		Type-correct	Ground truth match
Greedy search	1 3 5		
Non-greedy search	1 3 5		

Ground truth: 306 annotations in 47 fully annotated files

Exploring up to $7 \cdot |S|$ states

Effectiveness of Search

Strategy	Top- k	Annotations	
		Type-correct	Ground truth match
Greedy search	1	215 (70%)	194 (63%)
	3	230 (75%)	196 (64%)
	5	231 (75%)	198 (65%)
Non-greedy search	1	216 (71%)	195 (64%)
	3	178 (58%)	148 (48%)
	5	164 (54%)	141 (46%)

Ground truth: 306 annotations in 47 fully annotated files

Exploring up to $7 \cdot |S|$ states

Effectiveness of Search

Strategy	Top- k	Annotations	
		Type-correct	Ground truth match
Greedy search	1	215 (70%)	194 (63%)
	3	230 (75%)	196 (64%)
	5	231 (75%)	198 (65%)
Non-greedy search	1	216 (71%)	195 (64%)
	3	178 (58%)	148 (48%)
	5	164 (54%)	141 (46%)
Pyre Infer	—	106 (35%)	78 (25%)

Subsumes
practically
all types

Ground truth: 306 annotations in 47 fully annotated files

Exploring up to $7 \cdot |S|$ states

Limitations

- Type-correctness vs. soundness
- Limited type vocabulary
- Gradual type checking (and hence TypeWriter) is relatively slow

Summary: TypeWriter

Neural type prediction with search-based validation

- Probabilistic type prediction based on NL and PL information
- Ensure type correctness of added types via gradual type checker
- TypeWriter tool in use at Facebook

Why Does It Work?

Developers use **meaningful names**

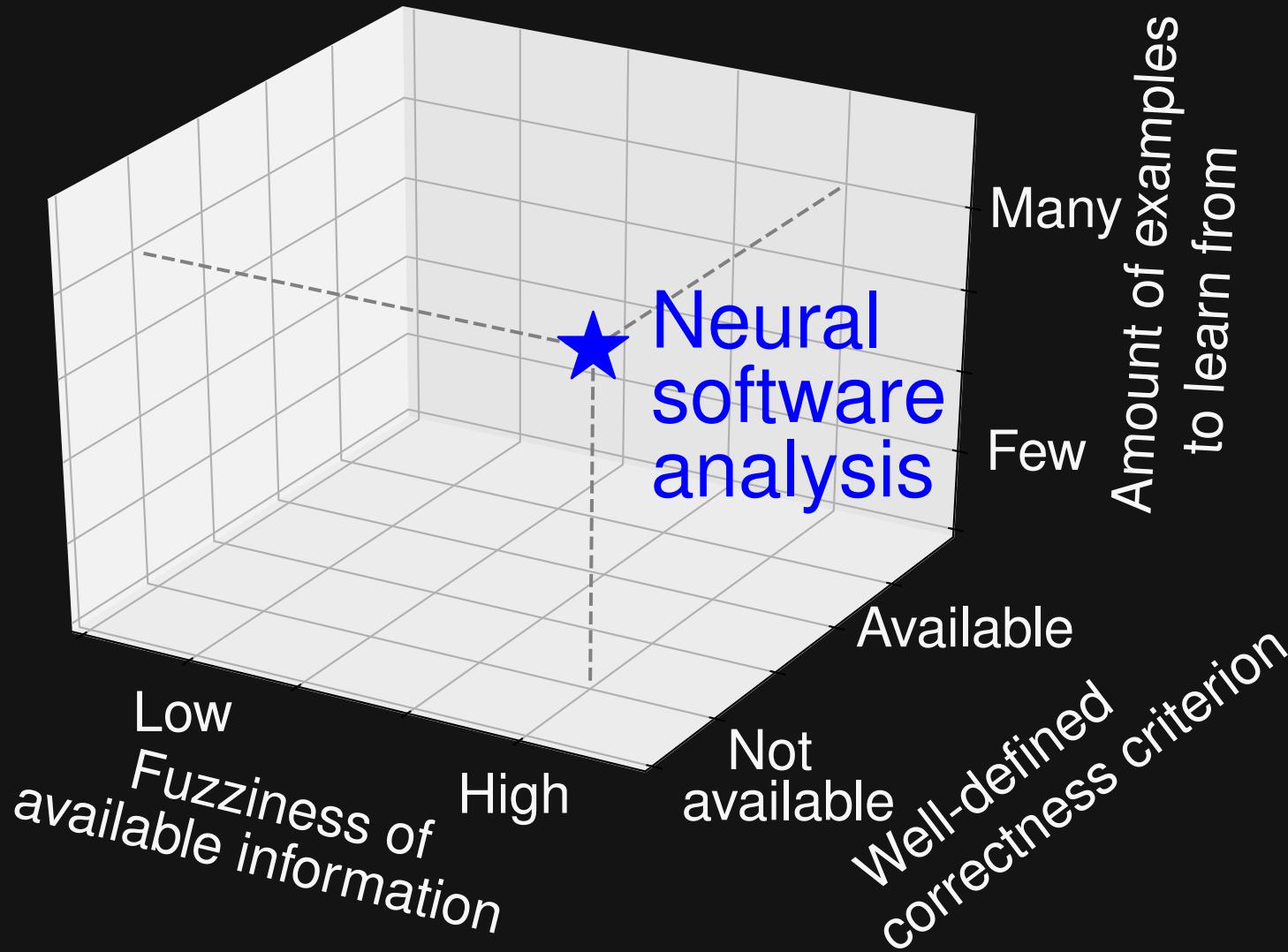
Source code is **repetitive**

Many programs available as **training data**

Probabilistic models + NL = ❤

Neural Software Analysis

When to (not) use it?



Other Interests

Neural software analysis:

- Human attention vs. neural models
- Semantic bug seeding
- Token-level code embeddings
- Ultra-large scale bug localization

Other topics:

- WebAssembly: Security and analysis
- Node.js security
- JavaScript test generation
- Program repair
- Quantum computing platforms

Conclusions

- Neural software analysis:
More and more code = ~~the problem~~
part of the solution
- Open challenges
 - Learning from runtime behavior
 - Clean datasets
 - Combining neural with traditional analysis