The Raise of Machine Learning Hyperparameter Constraints in Python Code

Ingkarat Rak-amnouykit
rakami@rpi.edu
Rensselaer Polytechnic Institute
USA

Ana Milanova
milanova@cs.rpi.edu
Rensselaer Polytechnic Institute
USA

Guillaume Baudart
guillaume.baudart@inria.fr
DI ENS, Ecole normale supérieure,
PSL University, CNRS, INRIA
France

Martin Hirzel
hirzel@us.ibm.com
IBM Research
USA

Julian Dolby
dolby@us.ibm.com
IBM Research
USA

ABSTRACT

Machine-learning operators often have correctness constraints that cut across multiple hyperparameters and/or data. Violating these constraints causes the operator to raise runtime exceptions, but those are usually documented only informally or not at all. This paper presents the first interprocedural weakest-precondition analysis for Python to extract hyperparameter constraints. The analysis is mostly static, but to make it tractable for typical Python idioms in machine-learning libraries, it selectively switches to the concrete domain for some cases. This paper demonstrates the analysis by extracting hyperparameter constraints for 181 operators from a total of 8 ML libraries, where it achieved high precision and recall and found real bugs. Our technique advances static analysis for Python and is a step towards safer and more robust machine learning.

CCS CONCEPTS

• **Theory of computation → Semantics and reasoning**; • **Software and its engineering → Functionality**.

KEYWORDS

Python, machine learning libraries, interprocedural analysis

ACM Reference Format:
Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. 2022. The Raise of Machine Learning Hyperparameter Constraints in Python Code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534400

1 INTRODUCTION

To use machine-learning (ML) operators, data scientists must configure their *hyperparameters*, usually via constructor arguments. For example, sklearn's `StandardScaler` operator has hyperparameters `with_mean` and `with_std`, and `LogisticRegression` has hyperparameters `dual`, `solver`, `penalty`, etc. [11]. Incorrect hyperparameter configurations raise exceptions, cause slowdowns, or yield suboptimal accuracy. But configuring hyperparameters correctly is often not easy due to *hyperparameter constraints*. For example, `StandardScaler` does not allow `with_mean==True` if input data is sparse, and `LogisticRegression` does not allow `dual==True` unless `solver=="liblinear"` and `penalty=="l2"`. We need a reliable formal specification of these constraints for dynamic precondition checks, static verifiers, or pruning automated hyperparameter search.

Unfortunately, it is difficult to find a reliable formal specification of hyperparameter constraints. Type annotations are insufficient: putting aside the fact that types are not yet widely adopted in Python and often wrong [33], they are also not expressive enough for constraints across multiple hyperparameters, or across hyperparameters and data. Hyperparameter tuning tools, such as auto-sklearn [15] or hyperopt-sklearn [23], come with search space specifications. But writing those specifications by hand is tedious and error-prone: for example, they take 25 KLOC of Python in auto-pandas [7]. Therefore, they often cut corners, making under-approximations (e.g., specifying only one of the types of a union) and over-approximations (e.g., missing constraints). This may be tolerable for search but is unacceptable for error checking.

One might be tempted to turn to natural-language documentation for hyperparameter constraints [6]. But even though popular packages like sklearn have high-quality documentation, this is at most semi-formal and not always reliable. The code may raise an undocumented exception. For example, using the techniques in this paper, we found that sklearn's `ExtraTreesClassifier` raised an exception if `bootstrap==False` and `oob_score==True`. But the documentation did not mention this constraint. As another example, our analysis found that `AffinityPropagation` raised an exception for sparse data, but the documentation said it handled sparse data. We submitted sklearn issues for both examples, and both were confirmed by the developers and fixed within days. The `ExtraTreesClassifier` fix updated the documentation to match the code and the `AffinityPropagation` fix updated the code to match the documentation.

This paper presents a static analysis for extracting hyperparameter constraints from code of ML operators. We focus on Python

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534400

and sklearn [11], the most widely-used ML framework today (as of January 2022, the "Used by" count of the sklearn GitHub repository was 299k, ahead of 174k for TensorFlow and 113k for PyTorch). Our analysis comprises four main components: 1) an intraprocedural weakest precondition analysis, 2) a call-graph analysis, 3) a soundness analysis using reference immutability, and 4) an interprocedural weakest precondition analysis. Our preconditions are logic formulas with constraints over at least two hyperparameters or hyperparameters and data. We can dynamically check these at the interface, which is friendlier than raising an exception from deep within the implementation. For better error messages, our analysis factors formulas to be easily associated with individual exceptions. We can also use these preconditions to prune search spaces for hyperparameter tuning. Moreover, we can envision using them for static verification of client code.

This paper tackles static analysis for Python, a problem that has received surprisingly little attention given the importance and widespread use of Python in data science programming. We set out to build an analysis for extracting hyperparameter constraints expecting to reuse existing results, only to discover that even classical analyses such as pointer analysis and call graph construction for Python remain open problems. The problem is difficult, due to the rich features of Python and their use in ML libraries. We build our analysis over the AST, applying classical ideas from Hoare logic and separation logic. We develop novel call graph construction and reference immutability analyses as well as a technique that switches between the analysis domain and the concrete domain to simplify analysis results (call graphs and weakest precondition formulas).

We ran our analysis on 181 ML operators from 8 ML libraries (122 sklearn operators plus operators from 7 other popular ML libraries). The analysis achieved 92.6% precision and 43.9% recall on input validation experiments, significantly improving over previous work on the problem. Our analysis also discovered issues in sklearn and imblearn, leading to 4 merged pull requests.

This paper makes the following contributions:

- The first interprocedural weakest precondition analysis for Python.
- A soundness analysis based on reference immutability.
- Formula simplification using concrete evaluation.
- Successful application to widely-used machine learning libraries.

Overall, we hope that our interface specifications make ML libraries more reliable and easier to use. Our extracted hyperparameter constraints are available with the submission.

2 OVERVIEW

This section illustrates our approach for extracting hyperparameter constraints using a weakest-precondition analysis of Python code. As a running example, Figure 1 shows an excerpt of the source code of the sklearn logistic regression operator.

2.1 Hyperparameters Constraints

A machine-learning *operator* is a class (L11). The *hyperparameters* correspond to the constructor arguments (L33). In sklearn, hyperparameters always have default values (L33) and are stored as instance attributes (L34-36). The class docstring (L12-32) specifies types, default values, and descriptions for hyperparameters.

```
1  from ..utils.multiclass import check_classification_targets
2  [...]
3
4  def _check_solver(solver, penalty, dual):
5    if solver not in ['liblinear', 'saga'] and penalty not in ('l2', 'none'):
6      raise ValueError(f"{solver} supports only 'l2' or 'none' penalties, got {penalty
           }.")
7    [...]
8    return solver
9
10
11 class LogisticRegression(LinearClassifierMixin, SparseCoefMixin,
       BaseEstimator):
12   """
13   Logistic Regression (aka logit, MaxEnt) classifier.
14
15   Parameters
16   ----------
17   penalty : {'l1', 'l2', 'elasticnet', 'none'}, default='l2'
18     Used to specify the norm used in the penalization. The 'newton-cg',
19     'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is
20     only supported by the 'saga' solver. If 'none' (not supported by the
21     liblinear solver), no regularization is applied.
22
23   solver : {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'
24     Algorithm to use in the optimization problem.
25     - 'newton-cg', 'lbfgs', 'sag' and 'saga' handle L2 or no penalty
26     - 'liblinear' does not support setting penalty='none'
27
28   l1_ratio : float, default=None
29     The Elastic-Net mixing parameter, with 0 <= l1_ratio <= 1. Only
30     used if penalty='elasticnet'.
31   [...]
32   """
33   def __init__(self, penalty='l2', solver='lbfgs', l1_ratio=None, [...]):
34     self.penalty = penalty
35     self.solver = solver
36     self.l1_ratio = l1_ratio
37     [...]
38
39   def fit(self, X, y, sample_weight=None):
40     solver=_check_solver(self.solver, self.penalty, self.dual)
41
42     if self.penalty == 'elasticnet':
43       if (not isinstance(self.l1_ratio, numbers.Number) or
44           self.l1_ratio < 0 or self.l1_ratio > 1):
45         raise ValueError(f"l1_ratio must be between 0 and 1; got {self.l1_ratio}"
               )
46
47     X, y = self._validate_data(X, y, [...])
48     check_classification_targets(y)
49     self.classes_ = np.unique(y)
50     [...]
```

Figure 1: Source excerpt of LogisticRegression operator from https://github.com/scikit-learn/scikit-learn/blob/15a949460/sklearn/linear_model/_logistic.py#L1012.

The description sometimes includes constraints between hyperparameters that must always hold, e.g., *'liblinear' does not support setting penalty='none'* (L26). But since these constraints are expressed in natural language, they are open to interpretation, possibly outdated, and challenging to extract for automatic tools [6]. For instance, in Figure 1, the constraint on `solver` and `penalty` is also rephrased in the description of `penalty`: *If 'none' (not supported by the liblinear solver)* (L20).

```
1  {"description": "penalty = 'elasticnet'
2     => is_number(l1_ratio) and (0 <= l1_ratio <= 1)",
3   "anyOf": [
4     {"type": "object",
5      "properties": {"penalty": {"not": {"enum": ["elasticnet"]}}}},
6     {"type": "object",
7      "properties":
8        {"l1_ratio": {"type": "number", "minimum": 0, "maximum": 1}}}]}
9
10 {"description": "solver in ['liblinear', 'saga']
11    or penalty in ['l2', 'none']",
12  "anyOf": [
13    {"type": "object",
14     "properties": {"solver": {"enum": ["liblinear", "saga"]}}},
15    {"type": "object",
16     "properties": {"penalty": {"enum": ["l2", "none"]}}}]}
```

Figure 2: JSON schemas of the two constraints extracted from the code of Figure 1

This paper proposes a static analysis to mine these constraints from the source code of the operator (as opposed to the docstring). For each `raise` exception statement, the analysis automatically extracts a weakest precondition that must hold to prevent that statement from executing. The analysis then encodes constraints as JSON Schema [31], a widely-supported and widely-adopted schema language. For instance, JSON Schema is the foundation of the Open API language for specifying REST APIs [30]. JSON Schema works well with Python and is expressive enough to encode complex constraints. For example, a recent AutoML tool relies on JSON Schema to specify ML hyperparameters including constraints [5].

2.2 Static Analysis

Our analysis comprises four sub-analyses: 1) intraprocedural, 2) call graph, 3) soundness analysis, and 4) interprocedural.

Intraprocedural Weakest Precondition Analysis. For each operator method, the analysis computes the precondition of each `raise` exception statement. For instance, in Figure 1, the `fit` method can raise a `ValueError` (L45). Analyzing backward the control flow that can cause this exception yields the following weakest precondition Q: `self.penalty = 'elasticnet'` $\Rightarrow$ `self.l1_ratio` $\in \mathbb{R} \wedge$ $0 \leq$ `self.l1_ratio` ≤ 1. The constraint is then compiled into the JSON schema shown in Figure 2 (L1-8). Hyperparameters correspond to object properties, and the JSON schema keyword `"anyOf"` expresses a disjunction (we encode $p \Rightarrow q$ as $\neg p \vee q$). The type condition `l1_ratio` $\in \mathbb{R}$ is translated to a JSON schema type `"number"`, and bounds are expressed with `"minimum"` and `"maximum"`.

Call Graph Construction. Sklearn often externalizes complex checks in dedicated functions. For instance, in Figure 1, function `_check_solver` (L4-8) ensures that its arguments `solver`, `penalty`, and `dual` respect a set of constraints. To collect constraints corresponding to function calls — such as `_check_solver` (L40) — this paper proposes a simple call graph analysis able to handle class hierarchy and functions imported from other modules. It relies on concrete evaluation to identify external calls that have no impact on the analysis results, e.g., library functions like `np.unique(y)` (L49).

Soundness Analysis. Unfortunately, a lot of Python code in practice has side effects, which can make the weakest precondition analysis unsound in general. An exception can occur even if the precondition holds if a statement modifies a location referenced in the precondition as a side effect. To mitigate this, we propose a soundness analysis to check if a precondition may be unsound. Given a statement S and a precondition Q, the analysis computes the set of locations *read* in the precondition ($read(Q)$) and the set of locations *modified* by the statement ($mod(S)$). The precondition is sound if these two sets are disjoint. In our running example, backwards analysis finds the precondition of L41 as Q: `self.penalty = 'elasticnet'` $\Rightarrow$ `self.l1_ratio` $\in \mathbb{R} \wedge 0 \leq$ `self.l1_ratio` ≤ 1. The call at L40 seemingly does not affect Q and the analysis propagates Q to the beginning of `fit`. However, the call still demands consideration — if it writes `self.l1_ratio` or `self.penalty`, then even if Q holds at the start of `fit`, that does not guarantee that Q holds after L40. Our soundness analysis computes mod(`_check_solver(...)`) = $\{\}$ and $read(Q) = \{$ `self.l1_ratio`, `self.penalty` $\}$. Since they do not intersect, we conclude that Q is a sound precondition at the start of `fit`. If Q holds, the exception at L45 will not be raised.

Interprocedural Weakest Precondition Analysis. The call graph analysis takes an (operator class, target method) pair and constructs the call graph rooted at the target method. Interprocedural analysis uses that call graph to trace back exceptions that occur in reachable methods in the call graph. It computes preconditions that must hold at the start of a target method, e.g., fit or predict, to prevent the exceptions from happening at runtime. Intraprocedural analysis on `_check_solver` returns the following weakest precondition: `solver` $\in$ `['liblinear', 'saga']` $\vee$ `penalty` $\in$ `['l2', 'none']`. Using the call graph, our analysis then maps the arguments `solver` and `penalty` to the corresponding hyperparameters `self.solver` and `self.penalty` (L40). Finally, it compiles the constraint to the JSON schema shown in Figure 2 (L10-15) for target method fit. The analysis also simplifies subformulas that depend on constants, such as default arguments in Python. This works by evaluating likely constants with the interpreter, as described in Section 6.

3 INTRA-PROCEDURAL ANALYSIS

Our intra-procedural weakest precondition analysis uses essentially standard backwards reasoning [3, 20, 24] and adapts it for Python. This section describes the core analysis and Section 5 describes our novel extension with soundness reasoning.

The analysis starts from a `raise` statement, then computes the precondition that must hold at the start of the enclosing function to prevent the exception. Each step of backward reasoning computes WP($stmt, Q_{\text{post}}$) $\mapsto Q_{\text{pre}}$: given a Python statement and a postcondition Q_{post}, our analysis returns a precondition Q_{pre}.

- WP(`raise E`, Q_{post}) handles Raise statements:

  return `False`

  At a blank raise statement, an exception is certain, so the precondition for not raising it is Q_{pre} = `False`.
- WP(`x = RHS`, Q_{post}) handles Assignment statements:

  return Q_{post}[`RHS/x`]

  The precondition Q_{pre} results from the substitution of left-hand side `x` with `RHS`.

```
1 # {(self.penalty=='elasticnet' ⟹ (isinstance(self.l1_ratio,Number) ∧ 0<=self.l1_ratio<=1)) ∧ (¬(self.penalty=='elasticnet') ⟹ True)
2 # ≡ self.penalty=='elasticnet' ⟹ (isinstance(self.l1_ratio,Number) ∧ 0<=self.l1_ratio<=1)}
3 if self.penalty == 'elasticnet':
4     # {((¬isinstance(self.l1_ratio,Number) ∨ self.l1_ratio<0∨self.l1_ratio>1) ⟹ False) ∧ (¬(¬isinstance(self.l1_ratio,Number) ∨ self.l1_ratio<0 ∨ self.l1_ratio>1) ⟹ True)
5     # ≡ isinstance(self.l1_ratio,Number) ∧ 0<=self.l1_ratio<=1}
6     if not isinstance(self.l1_ratio,Number) or self.l1_ratio<0 or self.l1_ratio>1:
7         # {False}
8         raise ValueError(f"l1_ratio must be between 0 and 1; got {self.l1_ratio}")
```

Figure 3: Inferring the weakest precondition for one of the exceptions raised by Logistic Regression.

- WP(`if E: Seq1 else: Seq2`, Q_{post}) handles If statements:

  $Q_{\text{pre1}} \longleftarrow$ WP(`Seq1`, Q_{post})
  $Q_{\text{pre2}} \longleftarrow$ WP(`Seq2`, Q_{post})
  return $(\texttt{E} \Longrightarrow Q_{\text{pre1}}) \wedge (\neg\, \texttt{E} \Longrightarrow Q_{\text{pre2}})$

  The precondition is standard. If the condition evaluates to `True`, then the weakest precondition of `Seq1` and Q_{post} must hold, otherwise, the weakest precondition of `Seq2` and Q_{post} must hold.
- WP(`for E1 in E2: Seq`, Q_{post}) handles For statements:

  if Q_{post} `== True` **then**
    $Q_{\text{body}} \longleftarrow$ WP(`Seq`, `True`)
    $Q_{\text{pre}} \longleftarrow$ `E1 ∈ E2` $\Longrightarrow Q_{\text{body}}$
    return Q_{pre}
  else
    return Q_{post}
  end if

  If the exception (recall that the analysis tracks a single `raise` statement) is nested in a For statement, then we guard the precondition with `E1 ∈ E2`. Variables in `E1` are bound at the For loop and the precondition Q_{body} may involve these variables. These variables are quantified in formula Q_{pre}, however, if Q_{pre} reaches the top-level `fit`, we ignore it because our target language, JSON Schema, does not support such functionality. On the other hand, if postcondition Q_{post} is not `True`, we simply propagate Q_{post} past the For statement. This can be unsound (the loop body may modify locations referenced in Q_{post}) and imprecise, i.e., stronger than the weakest precondition (the precondition is not qualified by the negated loop test), but works well in practice.
- WP(`other`, Q_{post}) handles Other statements:

  return Q_{post}

  Other statements are Python statements that do not match the syntax of the core subset specified above. These include `while`, `del`, `try`, and the rest of the Statement nodes specified by the Python AST. The code for Other propagates Q_{post} as-is, which is potentially unsound as the statement may modify locations referenced in Q_{post}, but see Section 5.

Figure 3 illustrates with the example from Figure 1 L42-L45. The figure uses standard Hoare logic notation with curly braces around logic formulas embedded between code statements. In our analysis, $\{Q_{\text{pre}}\}$ stmt $\{Q_{\text{post}}\}$ means Q_{pre} = WP(stmt, Q_{post}). Furthermore, Figure 3 indicates formula simplification by showing equivalent formulas, notated as $\{Q_{\text{unsimplified}} \equiv Q_{\text{simplified}}\}$.

4 CALL GRAPH CONSTRUCTION

Call graph construction for Python is non-trivial, complicated by imports, functions as first-class values, and complex features such as decorators and context managers. We are aware of a single publication on call graph construction in the literature, PyCG [36], and several GitHub repositories, most notably code2flow [35]. While both PyCG and code2flow produced quality call graphs, neither sufficed for our purposes — PyCG required that all files under analysis are specified at the command line, while our problem required crawling through the ML library and discovering imported classes and functions. Neither call graph handled inheritance in sklearn, which was crucial for interprocedural weakest precondition analysis. Unfortunately, call graph construction for Python remains an open problem — none of PyCG, code2flow, or our algorithm, built for the purposes of interprocedural weakest precondition, handles value flow or dynamic calls on receiver objects, e.g., `enc.fit()`.

4.1 Basic Algorithm

We propose a new call graph construction analysis that runs in seconds and achieves good accuracy for our purposes. We believe that it can be a used as a baseline when developing and benchmarking more complex and more accurate call graph construction algorithms. The analysis takes as input the full package (e.g., sklearn) and an (operator class, target method) pair, e.g. (`LogisticRegression`, `fit`) and produces the call graph resulting from a call of the target method on an operator class receiver. The analysis is a *name-based resolution* in its essence. It first crawls the package directory and creates two maps:

```
classTable    : package:class          → [base1,...,baseN]
functionTable : package:class:function → FunctionDef
```

The classTable is a map from the fully qualified class name to the list of (unqualified) names of base classes. Here *package* is the full path name of the file that contains the class definition and *class* is the unqualified name of the class. For example, class `LogisticRegression` in Figure 1 (L11) is represented in classTable as

```
'linear_model/_logistic.py:LogisticRegression' →
['LinearClassifierMixin','SparseCoefMixin','BaseEstimator']
```

The functionTable is a map from the fully qualified function name to the Python AST node corresponding to that function definition. For example, function `fit` (L39) is represented in functionTable as

```
'linear_model/_logistic.py:LogisticRegression:fit' →
ast.FunctionDef of fit
```

If the function has no enclosing class, then the class field in the key is the string `'None'`. For example,

```
'linear_model/_logistic.py:None:_check_solver' →
ast.FunctionDef of _check_solver
```

The algorithm constructs a call graph where each node is a (class, function) pair and where edges represent the calling relations. Starting at the pair (operator_class, entry_function), it visits all calls in entry_function's definition and adds new nodes and edges to the graph. When the algorithm adds a new (class, function) pair to the graph, it queues the corresponding function definition for processing. The process continues until no new nodes or edges are added.

The analysis uses the Python AST library which has representations of standard syntactic constructs. In addition to ast.FunctionDef, we make use of ast.Name(name), representing names such as variables, ast.Attribute(value,attr), which represents attribute access such as for example `np.unique`, or `self.estimator._validate_data`, and so on. The AST node type ast.Call(func,args,keywords) represents a function call, and analysis is difficult because node func can be arbitrarily complex.

Consider the call `check_classification_targets(X)` in Figure 1 L48, which is a Name call, the predominant kind of call. The analysis searches in functionTable and finds

```
utils/multiclass.py:None:check_classification_targets
```

It queues node

```
(None,utils/multiclass.py:None:check_classification_targets)
```

for processing (if it has not been processed already). The analysis also matches constructor calls, e.g, `LogisticRegression()`, calls through self, e.g., `self._validate_data(X, y, [...])` (L47), and package-qualified calls, e.g., `linear_model._logistic.check_solver()`.

The analysis has limitations, leaving some calls as *unresolved*. Most notably, there is no comprehensive value-flow analysis and expression calls such as `self.random_state_.shuffle(ordered_idx)` or `est.fit(X)`, as well as calls through function pointers, remain unresolved. Also, our target libraries sometimes outsource computation to Cython [8]. At this point, our analysis does not look at Cython files to try to find Cython class and function definitions. In our experiments, unresolved calls are split between (1) expressions and indirect calls and (2) Cython calls.

4.2 Concrete Evaluation

The most notable part of the analysis is the evaluation of external library calls directly in the Python interpreter. We separate imports into two categories, local imports and external imports. Local imports, typically relative imports, are sub-packages of the package under analysis and are in scope for the static analysis. External imports refer to separately installed dependencies and are out of scope. In typical machine-learning Python libraries, built-in calls and external calls, particularly calls to the `numpy` and `scipy` libraries, abound. When processing calls, the analysis encounters hundreds of built-in and external calls, which raises the question: *Are these calls unresolved due to limitations of the analysis, or are they built-in or external calls that generally have no impact on analysis results?*

We have a simple but general solution that filters out (certain) built-in and external calls. We evaluate the call with no arguments in the Python interpreter using its external import environment (picked up by a crawler). If evaluation causes a `TypeError` complaining of missing arguments, then we conclude that the call is an external call and the callee method is out of scope for the analysis. If evaluation causes another exception, e.g., a `NameError` exception, then the call remains unresolved. As an example, consider the call `np.unique(y)` (L49). When the analysis encounters this call, it tries its cases for Name, self call, etc. but fails to match. It then runs:

```
eval("import numpy as np; import ...; np.unique()")
```

which leads to the following error:

```
TypeError: unique() missing 1 required positional argument: 'ar'.
```

Our analysis concludes that `np.unique(y)` is an *external* call rather than an *unresolved* call. Note that sending the call as is will result in `NameError` due to the argument `y`. The distinction between *external* and *unresolved* calls has implications for the analysis, as it makes a distinction for what is largely "unachievable" for the analysis (external calls) and what is a potential limitation and room for improvement (unresolved).

Many calls are successfully resolved in the Python interpreter; e.g., 346 out of 524 calls in LogisticRegression are *external*.

5 SOUNDNESS ANALYSIS

Side-effects in Python make the weakest precondition analysis from Section 3 unsound in general. An exception can be raised even if the precondition holds if a statement modifies a location referenced in the precondition as a side effect. To mitigate this issue, we augment the core analysis from Section 3 with a *soundness flag*. Clearly, "soundness" is too strong a word given Python's complex dynamic nature. In addition, it is predicated upon assumptions about call graph correctness and behavior of library calls.

5.1 Soundness Flag

At each step of backward reasoning, the analysis now computes $\text{WP}(stmt, Q_{\text{post}}, S_{\text{post}}) \mapsto (Q_{\text{pre}}, S_{\text{pre}})$: given a Python statement, a postcondition Q_{post}, and a soundness flag S_{post}, our analysis returns a pair $(Q_{\text{pre}}, S_{\text{pre}})$ of a precondition Q_{pre} and its soundness flag S_{pre}. If the flag S_{pre} is true, then Q_{pre} is sound, i.e., if Q_{pre} holds at the corresponding program point, then the tracked exception is not raised. On the other hand, if S_{pre} is false, the exception may still be raised even if Q_{pre} holds. Below we describe the addition of the flag for Raise, Assignment, If, and Other statements.

- $\text{WP}(\texttt{raise E}, Q_{\text{post}}, S_{\text{post}})$ handles Raise statements:
  return (`False`, `True`)
  A blank `raise` makes an exception certain, so the precondition for not raising it is $Q_{\text{pre}} = \texttt{False}$ with soundness flag $S_{\text{pre}} = \texttt{True}$.

- $\text{WP}(\texttt{x=RHS}, Q_{\text{post}}, S_{\text{post}})$ handles Assignment statements:
  $Q_{\text{pre}} \leftarrow Q_{\text{post}}[\texttt{RHS/x}]$
  return $(Q_{\text{pre}}, S_{\text{post}} \wedge \mathit{mod}(\texttt{RHS}) \cap \mathit{read}(Q_{\text{pre}}) == \emptyset)$
  Our treatment follows the principles of separation logic [29]. If the set $\mathit{mod}(\texttt{RHS})$ of locations *modified* by RHS and the set $\mathit{read}(Q_{\text{pre}})$ of locations *read* by Q_{pre} are disjoint, then Q_{pre} is sound, meaning that Q_{pre} evaluates to true iff after the execution of `x=RHS`, Q_{post} evaluates to true. Otherwise, Q_{pre} is potentially unsound. In other words, if RHS has side-effects that modify some location referenced by Q_{pre}, then making Q_{pre} true before the execution of RHS does not necessarily make Q_{post} true after. Section 5.2 describes how to compute *mod* and *read* sets.

- WP(`if E: Seq1 else: Seq2`, Q_{post}, S_{post}) handles If statements:
  $(Q_1, S_1) \leftarrow \text{WP}(\texttt{Seq1}, Q_{\text{post}}, S_{\text{post}})$
  $(Q_2, S_2) \leftarrow \text{WP}(\texttt{Seq2}, Q_{\text{post}}, S_{\text{post}})$
  $Q_{\text{pre}} \leftarrow (\texttt{E} \Rightarrow Q_1) \wedge (\texttt{not E} \Rightarrow Q_2)$
  return $(Q_{\text{pre}}, S_1 \wedge S_2 \wedge \mathit{mod}(\texttt{E}) \cap \mathit{read}(Q_{\text{pre}}) == \emptyset)$
  It is sound if (1) Q_{post} is sound, (2) neither Seq1 nor Seq2 contain statements that invalidate the soundness, and (3) `E` has no effect on Q_1 or Q_2. In practice, `E` is almost always side-effect free.

- WP(`other`, Q_{post}, S_{post}) handles Other statements:
  return $(Q_{\text{post}}, S_{\text{post}} \wedge \mathit{mod}(\texttt{other}) \cap \mathit{read}(Q_{\text{post}}) == \emptyset)$
  The handling of `other` propagates Q_{post} as-is, however, it sets the soundness flag to `False` when `other` may interfere with the formula Q_{post}. Our work proposes systematic handling of loops and other intricate Python constructs (slices, generators, etc.). Instead of defining handlers for those constructs (e.g., loops are a known thorn in weakest precondition inference), we observe that they largely have no impact on analysis results, and therefore can be ignored. If the analysis can show that the weakest precondition formula and the statement do not interfere, then loops and calls and other intricate statements can be handled (i.e., ignored) safely.

We illustrate with a constraint in `ExtraTreeClassifier` that our analysis discovered. It was undocumented and we submitted a pull request which the developers immediately merged.

```
1  # {(self.bootstrap ∧ ¬ self.oob_score, True)}
2  n_samples_bootstrap = _get_n_samples_bootstrap(
3      n_samples=X.shape[0],max_samples=self.max_samples)
4  # {(self.bootstrap ∨ ¬ self.oob_score, True)}
5  self._validate_estimator()
6  # {(¬ self.bootstrap ∧ self.oob_score ⇒ False, True)
7  # ≡ (self.bootstrap ∧ ¬ self.oob_score, True)}
8  if not self.bootstrap and self.oob_score:
9      # {(False,True)}
10     raise ValueError("Out of bag score only available if bootstrap=True")
```

The exception in L10 yields the formula in L6-7. Propagating over the call in L5 entails computing mod(`self._validate_estimator()`) = { `self.base_estimator_` } and read(`self.bootstrap or not self.oob_score`) = { `self.bootstrap, self.oob_score` }. These sets do not intersect and the flag remains `True` as shown in L4. The call assignment in L2-3 entails computing mod(`_get_n_samples_bootstrap(...)`), which is { } because the call is side-effect free. This leads to the sound precondition `self.bootstrap or not self.oob_score` in L1. The actual code has about 40 additional lines until the beginning of the function.

5.2 $\mathit{mod}(S)$ and $\mathit{read}(Q)$

The analysis requires information about the set of locations *modified* by a statement (or statement sequence) as well as the set of locations *read* by a formula. In our implementation these are Python AST nodes, so both the *mod* and *read* analyses entail AST traversal.

At the core of *mod* is the known reference immutability analysis [21, 38]. It computes qualifiers for each variable and field in the program. A reference `x` is *read-only* if the object `x` refers to, or any of its transitive components, is not modified through `x`. E.g.,

- In `x.f=1` and in `x[0]=1`, `x` is not read-only (it is mutable).
- In `y=id(x); y[i]=1`, where `id` is the standard identity function that returns its argument, `x` is mutable.
- In `y=x.f; y.g=0`, `x` is mutable.

Reference immutability is different from object immutability. We may have a read-only reference `x` that points to o and there may be a mutable reference to o that would still allow mutation of o.

We have adapted the ReIm reference immutability type system [21, 27] for Python and inferred qualifiers for each variable and each field. We use our call graph from Section 4 as ReIm is inherently interprocedural. We assume that library calls are polymorphic with respect to immutability, i.e., they do not modify the argument; however, if the code modifies the left-hand-side of an external call assignment, then mutability is passed to the argument.

The novelty of our work, in addition to the integration with weakest precondition analysis, lies in the computation of what we call *fragment reference immutability*. Fragment reference immutability takes a Python statement and computes read-only or mutable qualifiers for *that single statement*. The example illustrates:

```
1  for e in lst:
2      x = m(z) # z is read–only as the call does not modify argument
3  z.append(...) # z is mutable here
```

In ReIm's "global" inference, `z` is mutable because of the mutation through append in L3. However, in fragment reference immutability of the For statement (L1-2), `z` is read-only, as the mutation occurs later in the code. Fragment reference immutability infers types exactly as ReIm does, except that at method calls it makes use of the argument and return types computed by ReIm's "global" inference.

$\mathit{mod}(S)$ contains three kinds of elements `x`, `x.*`, and `x.f`:

- `x` is a reference different from `self`. This means that statement S may write stack location `x`. If S contains an assignment `x = ...`, then `x` is placed into $\mathit{mod}(S)$.
- `x.*` means that S may write some part of the object that `x` refers to, however, the analysis cannot determine which part, i.e., which field. When fragment reference immutability infers `x` mutable, then `x.*` is placed into $\mathit{mod}(S)$. Here, `x` may be `self`. This is typically due to a call with a mutable argument, e.g., `m(x)`.
- `x.f` means that S may write precisely field `f` of `x`, i.e., there is an assignment `x.f = ...`. Since `x.*` takes precedence over `x.f`, we implement a minor extension to ReIm to track `self.f` locations that occur through calls through `self`. (At calls `self.m(...)` we propagate locations `self.f` into the caller instead of `self.*`.)

Thus, mod(`_get_n_samples_bootstrap(...)`) is { } because method `_get_n_samples_bootstrap` is read-only.

Similarly, $\mathit{read}(Q)$ contains three kinds of elements `x`, `x.*`, and `x.f`:

- `x` means that Q reads stack location `x`. Any reference variable that appears in Q is included in $\mathit{read}(Q)$.
- `x.*` means that Q may read some part of the object `x` refers to.
- `x.f` means that Q reads precisely field `f` of `x`.

Clearly, read(`self.bootstrap or not self.oob_score`) consists of {`self.bootstrap, self.oob_score`}.

Finally, $\mathit{mod}(S) \cap \mathit{read}(Q) == \emptyset$ evaluates to `True` if and only if none of the following is true:

(1) `x` is in both $\mathit{mod}(S)$ and $\mathit{read}(Q)$ for some `x`

(2) `x.f` or `x.*` is in $mod(S)$ and `y.g` or `y.*` is in $read(Q)$, for some `x`≠`self`, `y`≠`self`; `f` may be the same as `g`.
(3) `self.f` or `self.*` is in $mod(S)$ and `self.f` or `self.*` is in $read(Q)$

The above computation assumes that `self` is a unique reference on the current stack frame. On the other hand, it treats any `x.f` and `y.g`, where `x`≠`self` and `y`≠`self`, as potential aliases. We use the intersection symbol for notational convenience, even though this is a custom operation as defined here.

6 INTERPROCEDURAL WEAKEST PRECONDITION ANALYSIS

To handle runtime exceptions raised outside of a target function, we expand our analysis to be interprocedural by using the call graph described in Section 4 to trace exceptions along function call paths. The interprocedural analysis computes preconditions that must hold at the start of a *target function*, e.g., `fit` or `predict`, to prevent exceptions that could be raised by the target function itself or transitively reachable callees.

If a callee raises exceptions, their preconditions, after performing backward reasoning, become additional parts of the caller's preconditions. For each `raise` statement, intraprocedural analysis computes the precondition at the start of its enclosing function. Then we propagate that formula upward through the call graph until it reaches the target function. The number of preconditions of a target function equals the number of possible paths to all exceptions. For example, consider the excerpt of `LogisticRegression`'s `fit` target function from Figure 1 (L39). Its preconditions include the preconditions from its own `raise` (L45), those from callees `_check_solver` (L40) and `self._validate.data` (L47), as well as those from any other functions transitively reachable in the call graph.

6.1 Basic Algorithm

The analysis takes the transitive closure of a target function over the call graph, breaking cycles if they exist. Then it analyzes each function in reverse topological order. Preconditions of each function travel up the call graph via function calls. Each precondition is treated separately, whether it is from a function's own raise statement or from a function call. A precondition from a raise statement follows intraprocedural analysis. For a function call, each of its preconditions is substituted with appropriate function arguments. Then it follows the standard intraprocedural backward analysis from Section 3 until it reaches the entry point of the function.

At a function call, each precondition of the callee is substituted with the function call's arguments or the default values of formal arguments. For positional arguments, the analysis performs a straightforward substitution of arguments or default values for formals. Consider a call `call(arg1)` that the call graph has resolved to function `callee(p1,p2='default')` and the analysis has computed a list of preconditions $(Q^1_{\text{callee}}, \ldots, Q^n_{\text{callee}})$. IWP defines the computation of the *interprocedural weakest precondition*. IWP(`call(arg1)`, $(Q^1_{\text{callee}}, \ldots, Q^n_{\text{callee}})$) propagates the callee's preconditions $Q^1_{\text{callee}}, \ldots, Q^2_{\text{callee}}$ to the caller:

$$\textbf{return} \left(\begin{array}{l} Q^1_{\text{callee}}[\texttt{arg1/p1}][\texttt{'default'/p2}], \\ \ldots, \\ Q^n_{\text{callee}}[\texttt{arg1/p1}][\texttt{'default'/p2}] \end{array} \right)$$

```
1  # {¬ isinstance(accept_sparse,str) ⇒ ¬ accept_sparse is False}
2  def _ensure_sparse_format(spmatrix, accept_sparse, dtype, copy,
3                            force_all_finite, accept_large_sparse):
4      if isinstance(accept_sparse, str):
5          accept_sparse = [accept_sparse]
6      if accept_sparse is False:
7          raise TypeError('A sparse matrix was passed, but dense '
8                          'data is required. Use X.toarray() to '
9                          'convert to a dense numpy array.')
10
11 # {¬ isinstance(accept_sparse,str) ⇒ ¬ accept_sparse is False}
12 def check_array(array, accept_sparse=False, *, accept_large_sparse=True,
13                 dtype="numeric", order=None, copy=False,
14                 force_all_finite=True, ensure_2d=True,
15                 allow_nd=False, ensure_min_samples=1,
16                 ensure_min_features=1, estimator=None):
17     array = _ensure_sparse_format(array, accept_sparse=accept_sparse,
18                                   dtype=dtype, copy=copy,
19                                   force_all_finite=force_all_finite,
20                                   accept_large_sparse=accept_large_sparse)
21
22 class BaseEstimator:
23     # {¬ isinstance(check_params.get('accept_sparse',False),str)
24     # ⇒ ¬ check_params.get('accept_sparse',False) is False}
25     def _validate_data(self, X, y='no_validation', reset=True,
26                        validate_separately=False, **check_params):
27         X = check_array(X, **check_params)
28
29 class LogisticRegression(LinearClassifierMixin,
30                          SparseCoefMixin,
31                          BaseEstimator):
32     # {True}
33     def fit(self, X, y, sample_weight=None):
34         X, y = self._validate_data(X, y, accept_sparse='csr',
35                                    dtype=_dtype, order="C",
36                                    accept_large_sparse=solver!='liblinear')
```

Figure 4: Modified excerpt with simplified path from sklearn LogisticRegression `fit` function to one specific exception.

The above formula (and the rest of the section) elides the soundness flag for brevity. It is handled in the obvious way, e.g., given $(Q^1_{\text{callee}}, S^1_{\text{callee}})$ we have

$Q^1_{\text{caller}} \leftarrow Q^1_{\text{callee}}[\texttt{arg1/p1}][\texttt{'default'/p2}]$
$S^1_{\text{caller}} \leftarrow S^1_{\text{callee}} \wedge mod(\texttt{arg1}) \cap read(Q^1_{\text{caller}}) == \emptyset$
return $(Q^1_{\text{caller}}, S^1_{\text{caller}})$

6.2 Handling Keywords Arguments

Substitution becomes more complex with keyword arguments, particularly when `**kwargs` is involved, because `kwargs` is a Python dictionary that can be passed from a caller to callees deeply nested in the call chain and can be updated dynamically along the call chain. The analysis works in a reverse topological order of a call graph, making it sometimes impossible to determine at the call the actual keyword arguments stored in `**kwargs`, and which of the callee's formal arguments use their default values.

In the interest of saving space, we illustrate our handling of keywords arguments with an example. Figure 4 illustrates this situation. The values of keyword arguments to `check_array` (L27) are set at the call to `validate_data` (L34) and passed through in L27 via argument `check_params`. This is a modified source excerpt with

a simplified path from a target function, `LogisticRegression`'s fit, to one specific exception in the `_ensure_sparse_format` function.

Following the reverse topological order of the call graph, we start at `_ensure_sparse_format` (L2). The method's precondition Q (L1) is ¬ `isinstance(accept_sparse, str)` ⇒ ¬ `accept_sparse is False`. Function `check_array` contains a call to `_ensure_sparse_format`. The call does not involve `**kwargs` so the analysis performs a straightforward substitution. The precondition result is the same (L11). (As the name of the formal `accept_sparse` and the actual are the same.)

Next, in function `_validate_data`, a call to `check_array` involves `**kwargs` named `check_params`. At this point we are not able to determine which keyword arguments are in `**check_params`. In the general case, the analysis picks up the keyword arguments used in Q and substitutes each of them as in the example below. It uses the dictionary, e.g., `check_params` (still placeholder) and the default value as the default argument of `get`:

```
Q[check_params.get('accept_sparse',False)/accept_sparse]
  [check_params.get('accept_large_sparse',True)/accept_large_sparse]
    [check_params.get('dtype','numeric')/dtype], etc.
```

The resulting precondition for our example is shown at L23–24. The idea is that at the call to `validate_data` the analysis will substitute `check_params` with the concrete dictionary, and evaluate the calls to `get`; if the concrete dictionary specifies a value, then `get` would return that value, otherwise, it would return the default value.

Finally, in `fit`, the call `self._validate_data(..)` (L34) provides enough information to determine keyword arguments passed with `**check_params`. The analysis substitutes `check_params` with the concrete dictionary `{accept_sparse:'csr', dtype:_dtype, order:'C', accept_large_sparse:solver != 'liblinear'}`. The precondition reduces to ¬ `isinstance('csr',str)` => ¬ `'csr'is False`, which is of course `{True}` meaning that LogisticRegression's `fit` does not throw the `_ensure_sparse_format` exception.

6.3 Concrete Evaluation

The above example motivates our next idea. Due to use of default arguments, weakest precondition formulas can sometimes be fully evaluated. E.g., `isinstance('csr',str)` immediately evaluates to `True`. Such evaluation can significantly simplify formulas and improve scalability. Thus, during substitution, the analysis does concrete evaluation:

ast.Node(C1,C2)`[target/value]` handles substitution in a composite node ast.Node(C1,C2):

Q' ← ast.Node(C1`[target/value]`, C2`[target/value]`)
Q'' ← eval(Q')
return simplify(Q'')

eval(Q') tries to evaluate Q' in the interpreter using just the standard libraries `numpy` and `scipy` as the context of evaluation. If it evaluates to a constant, the analysis propagates this constant rather than Q'. Returning to the example, `isinstance('csr',str)` (the antecedent of formula in L23-24), which the interpreter immediately evaluates to `True`; we also have `not ('csr'is False)` (the consequent of formula in L23-24) which the interpreter resolves to `True` as well. simplify(Q'') performs standard simplification, most notably of implication formulas: `False` => Q becomes `True` and `True` => Q becomes Q.

A wrinkle in the handling of dictionaries is that some of the actual arguments in the dictionary might be expressions rather than constants. In that case, evaluation of the `get` expression will result in a NameError. Running `{'accept_sparse':'csr', 'dtype':_dtype, 'order':'C','accept_large_sparse':solver != 'liblinear'}. get('accept_sparse',False)` in the interpreter will raise a NameError because of `_dtype` and `solver`. That can be handled in either *simplify* or *eval* and we outline the handling in *eval*.

When creating the actual dictionary, whenever the actual keyword argument is *not* a constant, unparse the ast.Node into a string, then prepend a nonsensical string to it, and then store the string node into the dictionary. In our running example the ast.Name expression `_dtype` becomes the string `'?XYZ _dtype'` and the dictionary becomes `{'accept_sparse':'csr', 'dtype':"?XYZ _dtype", 'order':'C','accept_large_sparse':"?XYZ solver != 'liblinear'"}`. Calling `get('accept_sparse',False)` on the above receiver evaluates into `'csr'`. We can parse back the expression stored in the nonsensical string if it is needed later.

7 EXPERIMENTAL RESULTS

We run the IWP analysis on 8 ML libraries: sklearn (122 operators), as well as 7 independent libraries that are sklearn-compatible: XGBoost (2 operators), LightGBM (2 operators), imblearn (22 operators), category_encoders (17 operators), MAPIE (2 operators), metric_learn (13 operators), and sklearn_pandas (1 operator). Our analysis is general and can run on any library when provided an (operator class, target method) entry tuple. We choose the above libraries because they follow the sklearn convention and provide (operator class, `fit`) targets. We run the IWP analysis on these targets, a total of *181 operators*. When analyzing sklearn-compatible libraries we stop at the boundary with other libraries. Some of the sklearn-compatible libraries import sklearn functions, most often data validation functions and we do not reanalyze those functions.

Our evaluation considers four research questions:

RQ1 Is the IWP analysis effective at finding real issues?
RQ2 Is it effective for schema validation for ML operators and how does it compare to existing solutions?
RQ3 How well does the soundness analysis work?
RQ4 What is the impact of concrete evaluation and does the IWP analysis scale?

To answer RQ1, we track discrepancies between documentation and preconditions we infer from code. We reported 3 issues to the sklearn developers and 1 issue to the imblearn developers; all issues were fixed and merged into sklearn and imblearn.

To answer RQ2, we design a fuzzing mechanism that samples random configurations from an operator's JSON schema and checks for runtime exceptions during training, i.e., `fit` calls. We then test if the JSON schema returned by the IWP analysis stops invalid configurations while allowing valid ones. Our analysis achieves 92.6% precision and 43.9% recall and outperforms existing approaches.

To answer RQ3, we measure the percentage of inferred preconditions that are judged "sound" by our analysis across all 181 operators. 95.7% of all inferred preconditions were judged sound.

To answer RQ4, we report on the impact of the concrete evaluation simplifier from Section 6.3 and on analysis running times. The

analysis runs under 11 min per operator for all operators, 3 min on average.

7.1 RQ1: Does the analysis find real issues?

First, we discovered hyperparameter constraints that were not documented and reported them to the developers. As of now we have reported two such preconditions, one in sklearn and one in imblearn, and the developers issued PRs that added our preconditions to the documentation.

Second, having noticed inconsistencies in sklearn's sparsity checks, we ran an additional experiment. Operators in sklearn run data validation code as illustrated in Figure 4. As shown, validation code checks for sparsity of `X`, where the default is `accept_sparse=False`. Thus, by default, data validation raises an exception when a sparse `X` is passed. We isolated the following exception in `'sklearn/utils/validation.py:None:_ensure_sparse_format'`:

```
raise TypeError('A sparse matrix was passed, but dense data is
    required. Use X.toarray() to convert to a dense numpy array.')
```

and computed the preconditions up to each `fit` and each `predict` method in sklearn. This exception is guarded by multiple conditionals along lengthy call chains starting at `fit` or `predict`, and ending at `_ensure_sparse_format`. It fires up if `X` is sparse and `accept_sparse` is `False`. Analysis is tricky because there are many different ways the code can set `accept_sparse` to a value other than `False`; it is enabled by our novel interprocedural propagation. Figure 4 illustrates the call chain in `LogisticRegression` with simplified control flow.

Our analysis either reports (1) a constraint `not sp.issparse(X)` at the top of `fit/predict`, meaning that if the user passes a sparse matrix the exception is raised, or (2) `True`, as in Figure 4 L32, meaning that the exception is not raised with a sparse `X`. Case (2) is because the operator has set `accept_sparse` to an appropriate value. E.g., in Figure 4 L34, the operator sets `accept_sparse` to `'csr'`.

Case (1) indicates that the operator's `fit/predict` method does not accept sparse data and this ought to be specified in the docstring. If the analysis reports `not sp.issparse(X)` but the documentation states that the method accepts a sparse `X`, then there is definitely a bug, either a documentation bug or a code bug. Case (2) indicates that the operator's `fit/predict` may accept sparse `X`, as data validation code appears to accept sparse `X`. If the analysis reports `True`, but the docstring states that the method does not accepts sparse `X`, then this is not necessarily a bug, as the operator may indeed forbid sparse data due to some internal operational constraints.

We applied the analysis on the `fit` and `predict` methods in all sklearn operators. We detected 2 instances of case (1), one in `AffinityPropagation.predict` and one in `MeanShift.predict`. It appeared that in both cases the `predict` functions were meant to support sparse data and they did, but the data validation call forgot to pass an argument for `accept_sparse`, so it defaulted to `False`, which triggered the exception when a sparse `X` was passed. We reported the issue in `AffinityPropagation` to sklearn and suggested the following fix: `check_array(X, accept_sparse='csr')`. The developers issued and merged a pull request within days. Our PR for `MeanShift` led to a discussion among sklearn developers on whether this should be a documentation fix or a code fix, eventually settling on a documentation fix.

We detected 22 instances of case (2). 12 cases appear to be documentation bugs; documentation was not updated to reflect a code upgrade that added handling of sparse `X`. We have not reported the potential overspecification issues, but we plan to do so in the future.

And third, in addition to sklearn and imblearn, we have contributed PRs to IBM's Lale Auto-ML project [4], specifically improving the JSON schema constraints of 72 of sklearn's operators.

7.2 RQ2: Is the Analysis Effective for Schema Validation for ML Operators?

This experiment evaluates how well IWP hyperparameter constraints are able to stop invalid hyperparameter configurations, while allowing valid ones to proceed. They work as assertions at the entry-point methods. Catching invalid configurations early is important — imagine a pipeline where the first operator takes hours to run only to reach an invalid configuration of the second one.

Experimental Methodology. For each operator, we start with carefully crafted schemas for individual hyperparameters; these schemas capture constraints on individual hyperparameters, but do not capture constraints that involve multiple hyperparameters or data. They are JSON schemas from IBM's Lale Auto-ML project [4]. Sampling from the domain of these schemas, we generated 1,000 random hyperparameters configurations based on hyperparameters that are relevant to hyperparameter optimization. Then, we create a trial by calling the operator's `__init__` method with the hyperparameter configuration, then calling its `fit` method and checking for dynamic exceptions. We experiment with two kinds of datasets, dense and sparse, resulting in a total of 2,000 trials for each operator. A trial fails if an exception is raised and it passes otherwise.

The results from the dynamic exceptions are our ground truth. We translate weakest precondition constraints from the analysis into JSON schema [31] format and use schema validation to check the hyperparameter configuration against the schema. We define the categories for precision/recall as follows:

- A trial is a *true positive* if it fails and the hyperparameter configuration is invalid against the JSON schema (i.e., issues a warning) as well.
- A trial is a *false positive* if it passes and the hyperparameter configuration is invalid against the JSON schema.
- A trial is a *false negative* if it fails and the hyperparameter configuration is valid against the JSON schema.
- A trial is a *true negative* if it passes and the hyperparameter configuration is valid against the JSON schema.

Unfortunately, we only have the necessary carefully crafted JSON schemas for individual hyperparameter constraints for sklearn, XGBoost, and LightGBM. We report results on 101 operators: 99 out of 122 sklearn operators, and one each from XGBoost and LightGBM. For the remaining operators, either the trials for gathering the ground-truth exceeded the time limit or they required customized inputs that we could not craft.

How well does our analysis work? We report on 101 operators. We sum up trial results from all operators to calculate precision and recall. The analysis achieved **92.6% precision** and 43.9% recall

leading to an F_1-score of 59.6%. Our interprocedural analysis captures the vast majority of exceptions, in many cases interprocedural. For example, 7 weakest precondition constraints from 4 different functions of `LogisticRegression` precisely identify 1,406 hyperparameter configurations from the failed trials. As another example, in `PCA`, the analysis correctly identifies 400 true positives and 600 true negatives from the dense dataset; the analysis correctly rejects all sparse trials because PCA does not support sparse input and has an explicit exception in the main file that we infer and capture in JSON schema. (We note that there is an explicit `issparse` check in PCA that immediately rejects sparse inputs, rather than defaulting to the exception in validation code.) Interprocedural analysis is crucial for improving precision and recall, as the exceptions happen along call chains from `fit`.

The main source of false negatives (i.e., lower recall) is that weakest precondition expressions were beyond the expressive power of JSON schema. Clauses of preconditions that are inexpressible in JSON evaluate to `True`. As a result, in our experiments, schemas with inexpressible clauses are nearly always valid, i.e., accept everything. Concretely, about 15 of the operators in the dataset do not accept sparse input by specification and all 1000 trials with sparse datasets fail by throwing the exception in validation code that we investigated in RQ1. Unfortunately, the JSON schema for that exception always evaluates to `True` resulting in thousands of false negatives. We estimated that special handling of the schema for that exception would have increased the recall score by about 20 points, however, we have left this for future work as there were a non-trivial number of exceptions that IWP infers but the schema does not express. There are occasions when the IWP analysis misses exceptions as well, e.g., because of dynamic class loading. *On no occasion* did the analysis infer a precondition (expressible in JSON) that passed, but the corresponding exception fired. This is consistent with our soundness result (see section 7.3) — about 95% of our preconditions are sound.

How does our approach compare to other solutions? We consider two alternative approaches to extract machine-readable hyperparameter constraints: hand-written schemas [4] and documentation-extracted schemas [6]. The hand-written constraints are extracted by careful examination of the documentation. We use the same experimental methodology to compare IWP to these solutions.

Figure 5 shows an average F_1-score of 3 groups: the group with all 101 operators, the subgroup of 87 operators that have both Weakest Precondition and NL Docstrings schemas, and the subgroup of 39 operators that have all three schemas, including Hand-Written schema. Figure 5 shows that **our approach, which is automatic, performs significantly better than the hand-written constraints**.

On the subgroup of 39 operators, the precision and recall for IWP is 97.1% and 59.4% respectively, resulting in F_1-score of 73.7 as shown in Figure 5. Precision and recall for Hand-Written constraints is 39.0% and 22.6% respectively, resulting in the F_1-score of 28.6. The worse performance is mainly because hand-written constraints leave out constraints that appear in the code as exceptions but are missing from the documentation. Hand-written constraints also miss exceptions in imported modules. On a rare occasion, hand-written constraints reject hyperparameter configurations that are

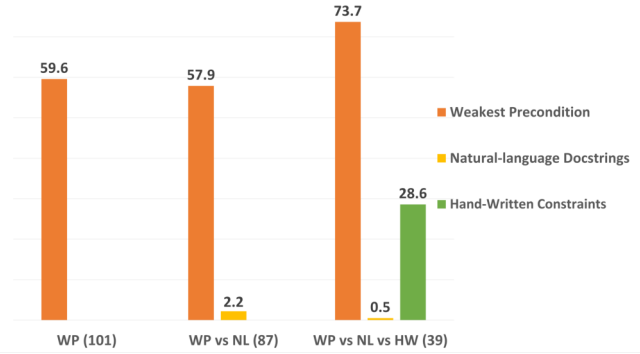

Figure 5: Average F_1-score of 3 groups; the group with all 101 operators (only IWP schemas), the subgroup of 87 operators (IWP and NL Docstrings schemas), and the subgroup of 39 operators (IWP, NL, and Hand-Written schemas).

specified in the documentation but do not exist in the code. For example, sklearn's `LinearSVC` states that the combination of `penalty='l1'` and `loss='hinge'` is not supported. However, no exception exists in the source code, resulting in 492 false positives.

Baudart et al. [6] automatically extract constraints for sklearn from natural-language documentation. While the technique works well for constraints on a single hyperparameter, it does not work as well for constraints on multiple hyperparameters or hyperparameters and data. Of the 101 operators, 87 operators have docstring-extracted schemas, but the technique only successfully extracts constraints that involve two or more hyperparameters or hyperparameters and data for 25 operators. Without these constraints, if a trial fails, its hyperparameter configuration is valid because there is nothing to validate against. This leads to a false negative and is the main reason of a low recall. Figure 5 shows that our weakest precondition analysis clearly outperforms the technique from [6].

7.3 RQ3: Does Soundness Analysis Work?

We ran the IWP analysis on all 181 operators starting at the target (operator,`fit`). Recall that this turns each transitively reachable exceptions into preconditions at the top of the operator's `fit` function, where each precondition is accompanied by a soundness flag. In summary, **95.5% of the sklearn preconditions, across all 122 operators, were inferred sound**. For the remaining 7 libraries, **97.0% of the preconditions were inferred sound**.

One wrinkle is that initially only about 35% of the sklearn preconditions were judged sound, which was highly surprising. Upon a closer look, the low soundness result across all sklearn operators was due to a *single For loop* in the shared data validation code, specifically in `check_array`. There are about 10 exceptions raised in `check_array` or its callees and each gives rise to 3-4 distinct preconditions at the top level because there are multiple paths from `fit` to `check_array`. Thus, the single source of imprecision (i.e., the For loop) propagates towards a large number of exceptions. We ran an experiment and excluded exceptions in `check_array` and, as expected, 94% of the inferred preconditions were sound.

The offending For loop assigned a local, `has_pd_integer_array`, which also appeared in the postcondition formula, thus rendering

Table 1: Running times and file sizes of some operators, with and without the concrete evaluation (CE). The files store AST formulas of weakest preconditions in .pkl format.

	Time (seconds)		File Size (MB)	
	CE	no CE	CE	no CE
AdaBoostClassifier	2,564.86	3,005.27	4.54	301.94
AdaBoostRegressor	2,197.82	3,196.64	4.68	302.09
BaggingClassifier	2,645.32	5,607.13	33.15	419.20
ColumnTransformer	405.58	424.61	2.35	46.71
ExtraTreesClassifier	2,828.72	3,969.06	131.19	727.10
StandardScaler	4,204.84	38,731.67	110.39	1,479.45
TfidfVectorizer	0.90	1.58	0.03	0.03
VotingClassifier	1.02	1.85	0.04	0.04

the precondition formula unsound. Fortunately, we were able to replace (manually) the For statement with an equivalent If statement that assigned `has_pd_integer_array` accordingly. The If statement is equivalent (under some mild assumptions) in the following sense: WP(`For`,Q) evaluates to true for the exact same values that WP(`If`,Q) evaluates to true (recall that WP(`For`,Q) is not computable in general; our analysis limits the impact of loops by reasoning about what variables are modified within a loop). Then we could analyze the full code with the If statement, propagating the postcondition into a sound precondition. Soundness has implications for schema validation (Section 7.2) — an unsound precondition may pass validation while the exception still fires at runtime and this never happened in our experiments.

7.4 RQ4: Does the Analysis Scale?

Our analysis, like all analyses in this space, suffers from path explosion. This is because exceptions are nested deeply into control-flow paths that span multiple functions and if-then-else statements. Yet the analysis still scales and computes a solution for each operator.

Concrete evaluation is crucial for scalability. As an ablation study, we measure the running time of the analysis and the size of the result file with and without using concrete evaluation. We can disable concrete evaluation by commenting out line $Q' \leftarrow \text{eval}(Q)$ of Section 6.3 (no CE). The result file is a Python pickle file (.pkl format) containing AST formulas of the operator's weakest precondition constraints. Table 1 shows the comparison on 8 representative sklearn operators. As expected, the difference in running time becomes more pronounced as precondition formulas become larger. Overall, concrete evaluation speeds up the analysis and simplifies constraints' complexity significantly. Speedup ranges from 1.0x to 9.2x for an average of 2.5x. File size reduction ranges from 5.5x to 66.5x for an average of 23.1x. Still, notice that in some cases the analysis takes about an hour, e.g., StandardScaler in Table 1. To further improve scalability, we apply a *pruning heuristic* that removes formulas that are excessively large, specifically, 200 implications or more. Such formulas are uninterpretable and not useful for our client analyses. All client analyses presented earlier make use of the preconditions computed with CE and pruning on.

With CE and pruning the analysis completes in **under 11 minutes on all operators with an average about 3 minutes for the 181 operators**. We run the experiments on a Windows machine with Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz.

8 RELATED WORK

Our non-archival preliminary work [34] introduces the idea of using weakest precondition inference for schema validation for ML operators; it presents an intra-procedural analysis and evaluation of schema validation on 45 sklearn operators. This work builds upon and significantly extends [34] with (1) novel call graph and *interprocedural* weakest precondition analyses (motivated by deficiencies of the intraprocedural analysis), and (2) *soundness analysis* to reason about quality of inferred preconditions. Additionally, we expand the experimental evaluation and apply IWP to specification inference, leading to new successful pull requests.

While backward reasoning and, more generally, verification condition generation have a long history, e.g., [3, 16] among other works, as far as we know, we are the first to apply these technique on Python, whose rich dynamic semantics notoriously complicate static analysis. Our work demonstrates scalability and applicability of these powerful techniques on real-world Python code.

Concrete evaluation in the context of a static analysis has some similarity to type-level computation theory [13, 22]. Kazerounian et al. [22] evaluate certain Ruby library calls towards proving type safety of database queries in Ruby programs. Hirzel et al. perform pointer analysis on Java that is sound modulo classes loaded and reflection called up to that point in the program execution, adding points-to relations when relevant concrete evaluations occur [19]. Our work applies evaluation in the Python interpreter to improve call graph construction and to simplify weakest precondition formulas. It is also related to concolic testing [26], as that evaluates SMT formulas to find inputs that improve coverage. PyExZ3 [2] and PyCT [12] are dynamic concolic testers for Python functions. In contrast, our analysis is static, includes reasoning about soundness, and it is interprocedural.

In general, work on static analysis for Python is scarce. Ariadne [14] explores static analysis of machine-learning libraries and outlines challenges to traditional static analysis techniques and Monat et al. [28] present type analysis via abstract interpretation; we focus on the specific problem of extracting hyperparameter constraints. There is a body of work on type inference for Python, including [25], [39], and [18]. Recent work explores machine-learning-based type inference, including [1] and [32]. Our work focuses on inference of deeper semantic properties such as hyperparameter and data constraints. iComment for C [37] and jDoctor for Java [9] have similar goal to ours — reconciling documentation with code and identifying issues with either of them.

Data validation for ML pipelines is an important problem. Breck et al. present a system for detection of anomalies in data fed to machine-learning pipelines [10]. Habib et al. check data flowing through ML pipelines using JSON subschema checks [17]. Our work analyzes parts of the Python code that performs data validation and checks whether it conforms to the data constraints specified in the documentation. Hyperparameter specifications, including constraints, are important for automated machine learning (Auto-ML). For example, the Auto-ML tools auto-sklearn [15] and hyperopt-sklearn [23] come with hand-written hyperparameter schemas. Lale

also has schemas extracted from docstrings [6]. In contrast, our paper is the first to show how to extract them via static analysis of the code.

9 CONCLUSIONS

This paper presents an interprocedural static analysis for extracting weakest preconditions from Python. We automatically transform the analysis results to JSON schemas suitable for validation as well as for automatic tuning of machine-learning hyperparameters. We add reasoning about soundness using reference immutability, following the principles of separation logic. We have successfully applied the analysis on 181 popular ML operators.

ACKNOWLEDGMENTS

We thank the ISSTA 2022 reviewers for their constructive feedback. We are especially grateful to the reviewers for pointing out the inconsistent interpretation of precision and recall in our submission. The first and second authors are supported by NSF grant #1814898.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Conference on Programming Language Design and Implementation (PLDI)*. 91–105. https://doi.org/10.1145/3385412.3385997

[2] Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Dependable Software Systems Engineering*. Vol. 40. IOS Press, 26–41.

[3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Symposium on Formal Methods for Components and Objects (FMCO)*. 364–387. https://doi.org/10.1007/11804192_17

[4] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2020. Lale: Consistent Automated Machine Learning. In *KDD Workshop on Automation in Machine Learning (AutoML@KDD)*. https://arxiv.org/abs/2007.01977

[5] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, Avraham Shinnar, and Jason Tsay. 2021. Pipeline Combinators for Gradual AutoML. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf

[6] Guillaume Baudart, Peter Kirchner, Martin Hirzel, and Kiran Kate. 2020. Mining Documentation to Extract Hyperparameter Schemas. In *ICML Workshop on Automated Machine Learning (AutoML@ICML)*. https://arxiv.org/abs/2006.16984

[7] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3360594

[8] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science and Engineering (CISE)* 13, 2 (2011), 31–39. https://doi.org/10.1109/MCSE.2010.118

[9] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*. 242–253. http://doi.acm.org/10.1145/3213846.3213872

[10] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *Conference on Systems and Machine Learning (SysML)*. https://mlsys.org/Conferences/2019/doc/2019/167.pdf

[11] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API Design for Machine Learning Software: Experiences from the scikit-learn Project. https://arxiv.org/abs/1309.0238

[12] Yu-Fang Chen, Wei-Lun Tsai, Wei-Cheng Wu, Di-De Yen, and Fang Yu. 2021. PyCT: A Python Concolic Tester. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 38–46. https://doi.org/10.1007/978-3-030-89051-3_3

[13] Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. In *Conference on Programming Language Design and Implementation (PLDI)*. 122–133. https://doi.org/10.1145/1809028.1806612

[14] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Workshop on Machine Learning and Programming Languages (MAPL)*. 1–10. http://doi.acm.org/10.1145/3211346.3211349

[15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Conference on Neural Information Processing Systems (NIPS)*. 2962–2970. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning

[16] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Conference on Programming Language Design and Implementation (PLDI)*. 234–245. https://doi.org/10.1145/543552.512558

[17] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *International Symposium on Software Testing and Analysis (ISSTA)*. 620–632. https://doi.org/10.1145/3460319.3464796

[18] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Conference on Computer Aided Verification (CAV)*. 12–19. https://doi.org/10.1007/978-3-319-96142-2_2

[19] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. 2007. Fast Online Pointer Analysis. *Transactions on Programming Languages and Systems (TOPLAS)* 29, 2 (April 2007). https://doi.org/10.1145/1216374.1216379

[20] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[21] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReImm & ReImInfer: checking and inference of reference immutability and method purity. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 879–896. https://doi.org/10.1145/2398857.2384680

[22] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. 2019. Type-Level Computations for Ruby Libraries. *CoRR* abs/1904.03521 (2019). https://doi.org/10.1145/3314221.3314630

[23] Brent Komer, James Bergstra, and Chris Eliasmith. 2014. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. In *Python in Science Conference (SciPy)*. 32–37. http://conference.scipy.org/proceedings/scipy2014/komer.html

[24] K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inform. Process. Lett.* 93, 6 (2005), 281–288. https://doi.org/10.1016/j.ipl.2004.10.015

[25] Eva Maia, Nelma Moreira, and Rogério Reis. 2011. A Static Type Inference for Python. In *Workshop on Dynamic Languages and Applications (DYLA)*. http://scg.unibe.ch/download/dyla/2011/dyla11_submission_3.pdf

[26] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *International Conference on Software Engineering: Companion (ICSE-C)*. https://doi.org/10.1109/ICSE.2007.41

[27] Ana Milanova. 2018. Definite Reference Mutability. In *European Conference for Object-Oriented Programming (ECOOP)*, Todd D. Millstein (Ed.). 25:1–25:30. https://doi.org/10.4230/LIPIcs.ECOOP.2018.25

[28] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29. https://doi.org/10.4230/DARTS.6.2.11

[29] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Workshop on Computer Science Logic (CSL)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

[30] OpenAPI Initiative. 2014. OpenAPI Specification (fka Swagger RESTful API Documentation Specification). https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md.

[31] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *International Conference on World Wide Web (WWW)*. 263–273. https://doi.org/10.1145/2872427.2883029

[32] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Symposium on the Foundations of Software Engineering (FSE)*. 209–220. https://doi.org/10.1145/3368089.3409715

[33] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *Dynamic Languages Symposium (DLS)*. 57–70. https://doi.org/10.1145/3426422.3426981

[34] Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. 2021. Extracting hyperparameter constraints from code. In *ICLR Workshop on Security and Safety in Machine Learning Systems (SecML@ICLR)*. https://aisecure-workshop.github.io/aml-iclr2021/papers/18.pdf

[35] Scott Rogowski. 2021. *code2flow*. Retrieved January 26, 2021 from https://github.com/scottrogowski/code2flow

[36] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *International Conference on Software Engineering (ICSE)*. 1646–1657. https://arxiv.org/abs/2103.00587

[37] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or Bad Comments? */. In *Symposium on Operating Systems Principles (SOSP)*. 145–158. https://doi.org/10.1145/1294261.1294276

[38] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 211–230. https://doi.org/10.1145/1094811.1094828

[39] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Symposium on the Foundations of Software Engineering (FSE)*. 607–618. https://doi.org/10.1145/2950290.2950343