The raise of machine learning hyperparameter constraints in Python code
Ingkarat Rak-Amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, Julian Dolby

To cite this version:
Ingkarat Rak-Amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, Julian Dolby. The raise of machine learning hyperparameter constraints in Python code. ISSTA 2022 - 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Jul 2022, Virtual, South Korea. 10.1145/3533767.3534400 . hal-03891774

HAL Id: hal-03891774
https://hal.science/hal-03891774
Submitted on 9 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
The Raise of Machine Learning Hyperparameter Constraints in Python Code

Inkarat Rak-amnouykit
rkami@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:

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 [34], 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
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.

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 OVERVIEW

This section describes 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.
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' ⇒ self.l1_ratio ∈ ℝ ∧ 0 ≤ 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 ⇒ q as ¬p ∨ q). The type condition l1_ratio ∈ 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' ⇒ self.l1_ratio ∈ ℝ ∧ 0 ≤ 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 ∈ ['liblinear', 'saga'] V penalty ∈ ['12', '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, Qpost) ⇒ Qpre: given a Python statement and a postcondition Qpost, our analysis returns a precondition Qpre.

- WP(raise E, Qpost) handles Raise statements:
  ```python
  return False
  ```
  At a blank raise statement, an exception is certain, so the precondition for not raising it is Qpre = False.
- WP(x = RHS, Qpost) handles Assignment statements:
  ```python
  return Qpost[RHS/x]
  ```
  The precondition Qpre results from the substitution of left-hand side x with RHS.
The precondition is standard. If the condition evaluates to True, then the weakest precondition of \texttt{seq1} and \texttt{post} must hold, otherwise, the weakest precondition of \texttt{seq2} and \texttt{post} must hold.

**WP**(\texttt{for \texttt{E1} in \texttt{E2}: \texttt{Seq2} \texttt{post}) handles \texttt{For} statements:

- \texttt{if post == True then}
  \texttt{Qbody} \texttt{= WP(Seq2, True)}
  \texttt{Qpre} \texttt{= E1 in E2 => Qbody}
- \texttt{return Qpre}

  *If the exception (recall that the analysis tracks a single raise statement) is nested in a For statement, then we guard the precondition with \texttt{E1 in E2}. Variables in \texttt{E1} are bound at the For loop and the precondition \texttt{Qbody} may involve these variables. These variables are quantified in formula \texttt{Qpre}, however, if \texttt{Qpre} reaches the top-level \texttt{fit}, we ignore it because our target language, JSON Schema, does not support such functionality.*

  *On the other hand, if postcondition \texttt{Qpost} is not True, we simply propagate \texttt{Qpost} past the For statement. This can be unsound (the loop body may modify locations referenced in \texttt{Qpost}) 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**(\texttt{other, Qpost}) handles \texttt{Other} statements:

- \texttt{return Qpost}

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

Piece 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, \{\texttt{Qpre}\} \texttt{stmt} \{\texttt{Qpost}\} means \texttt{Qpre = WP(stmt, Qpost)}. Furthermore, Figure 3 indicates formula simplification by showing equivalent formulas, notated as \{\texttt{Qunsimplified} \equiv \texttt{Qsimplified}\}.

### 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 [37], and several GitHub repositories, most notably code2flow [36]. 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., \texttt{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 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., \{\texttt{LogisticRegression}, \texttt{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:

- \texttt{classTable : package: class} \rightarrow \texttt{[base1,...,baseN]}
- \texttt{functionTable : package: class:function} \rightarrow \texttt{FunctionDef}

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

\begin{quote}
  \begin{verbatim}
  'linear_model/_logistic.py:LogisticRegression' \rightarrow
  ['LinearClassifierMixin', 'SparseCoefMixin', 'BaseEstimator']
  \end{verbatim}
\end{quote}

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

\begin{quote}
  \begin{verbatim}
  'linear_model/_logistic.py:LogisticRegression.fit' \rightarrow
  ast.FunctionDef of fit
  \end{verbatim}
\end{quote}

If the function has no enclosing class, then the class field in the key is the string \texttt{'None'}. For example,

\begin{quote}
  \begin{verbatim}
  'linear_model/_logistic.py:None_check_solver' \rightarrow
  ast.FunctionDef of _check_solver
  \end{verbatim}
\end{quote}
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

    def check_classification_targets(X)
    
    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(orderd_ids) or est.fit(X), as well as calls through function pointers, remain unresolved. Also, our target libraries sometimes outsource computation to Cython [5]. 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 WP(stmt, Qpost, Spost) → (Qpre, Spre): given a Python statement, a postcondition Qpost, and a soundness flag Spost, our analysis returns a pair (Qpre, Spre) of a precondition Qpre and its soundness flag Spre. If the flag Spre is true, then Qpre is sound, i.e., if Spre holds at the corresponding program point, then the tracked exception is not raised. On the other hand, if Spre is false, the exception may still be raised even if Qpre holds. Below we describe the addition of the flag for Raise, Assignment, If, and Other statements.

- WP(r=raise E, Qpost, Spost) handles Raise statements:
  ```python
  return (False, True)
  ```
  A blank raise makes an exception certain, so the precondition for not raising it is Qpre = False with soundness flag Spre = True.

- WP(x=RHS, Qpost, Spost) handles Assignment statements:
  ```python
  Qpre ← Qpost[RHS/x]
  return (Qpre, Spost ∧ mod(RHS) ∧ read(Qpre) == ∅)
  ```
  Our treatment follows the principles of separation logic [29]. If the set mod(RHS) of locations modified by RHS and the set read(Qpre) of locations read by Qpre are disjoint, then Qpre is sound, meaning that Qpre evaluates to true iff after the execution of x=RHS, Qpost evaluates to true. Otherwise, Qpre is potentially unsound. In other words, if RHS has side-effects that modify some location referenced by Qpre, then making Qpre true before the execution of RHS does not necessarily make Qpost true after. Section 5.2 describes how to compute mod and read sets.
• WP(if E: Seq1 else: Seq2, Qpost, Spost) handles if statements:
  (Q1, S1) ← WP(Seq1, Qpost, Spost)
  (Q2, S2) ← WP(Seq2, Qpost, Spost)
  Qpre ← (E ⇒ Q1) ∧ (¬ E ⇒ Q2)
  return (Qpre, S1 ∧ S2 ∧ mod(E)∩read(Qpre) == ∅)

It is sound if (1) Qpost is sound, (2) neither Seq1 nor Seq2 contain statements that invalidate the soundness, and (3) E has no effect on Q1 or Q2. In practice, E is almost always side-effect free.

• WP(other, Qpost, Spost) handles other statements:
  return (Qpost, Spost ∧ mod(other)∩read(Qpost) == ∅ )

The handling of other propagates Qpost as-is, however, it sets the soundness flag to False when other may interfere with the formula Qpost. 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 ExtraFreeClassifier 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.base_estimator, 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 mod(S) and 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, 39]. 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[1]=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 Relm 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 Relm 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:

```python
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 Relm’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 Relm does, except that at method calls it makes use of the argument and return types computed by Relm’s “global” inference.

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 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 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 Relm 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, 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 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, mod(S) ∩ read(Q) == ∅ evaluates to True if and only if none of the following is true:

1. x is in both mod(S) and read(Q) for some x
(2) x.f or x.* is in \(\text{mod}(S)\) and y.g or y.* is in \(\text{read}(Q)\), for some \(x \neq y\). If \(x \neq y\) and \(x \neq \text{self}\), \(x.f\) may be the same as \(g\).

(3) self.f or self.* is in \(\text{mod}(S)\) and self.f or self.* is in \(\text{read}(Q)\).

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 \(x \neq \text{self}\) and \(y \neq \text{self}\), 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., \(\text{fit}\) or \(\text{predict}\), 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 callee’s preconditions. For each \(\text{raise}\) statement, interprocedural 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 \(\text{LogisticRegression}\)'s \(\text{fit}\) target function from Figure 1 (L39). Its preconditions include the preconditions from its own \(\text{raise}\) (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 \(\text{call}(\text{arg})\) that the call graph has resolved to function \(\text{callee(p1,p2='default')}\) 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(caller), \((Q^1_{\text{callee}}, \ldots, Q^n_{\text{callee}})\) propagates the caller’s preconditions \((Q^1_{\text{callee}}, \ldots, Q^n_{\text{callee}})\) to the caller:

\[
\begin{align*}
\text{return } (Q^1_{\text{caller}}, Q^n_{\text{caller}}) \text{ to the caller:} \\
&Q^1_{\text{caller}}[\text{arg}1/p1][\text{’default’}/p2], \\
&\quad \ldots, \\
&Q^n_{\text{caller}}[\text{arg}1/p1][\text{’default’}/p2].
\end{align*}
\]

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 \((Q^1_{\text{caller}}, S^1_{\text{callee}})\) we have

\[
\begin{align*}
Q^1_{\text{caller}} &\leftarrow Q^1_{\text{callee}}[\text{arg}1/p1][\text{’default’}/p2], \\
S^1_{\text{caller}} &\leftarrow S^1_{\text{callee}} \land \text{mod(\text{arg})} \land \text{read}(Q^1_{\text{caller}}) == \emptyset \\
\text{return } (Q^1_{\text{caller}}, S^1_{\text{callee}}).
\end{align*}
\]

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

---

**Figure 4:** Modified excerpt with simplified path from sklearn LogisticRegression fit function to one specific exception.
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 \( \neg \text{isinstance}(\text{accept_sparse}, \text{str}) \Rightarrow \neg \text{accept_sparse} \) 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:

\[
Q[\text{check_params}.\text{get}(\text{accept_sparse}, \text{False})/\text{accept_sparse}]
\]

The 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 will 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 \( \neg \text{isinstance}(\text{csr}, \text{str}) \Rightarrow \neg \text{csr} \) 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, \( \text{isinstance}(\text{csr}, \text{str}) \) immediately evaluates to True. Such evaluation can significantly simplify formulas and improve scalability. Thus, during substitution, the analysis does concrete evaluation:

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

\[
Q' \leftarrow \text{ast.Node}(\text{C1[target/value], C2[target/value]})
\]

\[
Q'' \leftarrow \text{eval}(Q')
\]

\[
\text{return simplify}(Q'')
\]

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, \( \text{isinstance}(\text{csr}, \text{str}) \) (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: \( \text{False} \Rightarrow Q \) becomes True and True \( \Rightarrow Q \) becomes False.

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 constants. In that case, evaluation of the expression will result in a NameError.

Running (’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.
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, a validation code checks for sparsity of \( X \), where the default is \( \text{accept\_sparse} = \text{False} \). 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 \text{fit} and each \text{predict} method in sklearn. This exception is guarded by multiple conditionals along lengthy call chains starting at \text{fit} or \text{predict}, and ending at \_ensure\_sparse\_format. It fires up if \( X \) is sparse and \( \text{accept\_sparse} = \text{False} \). Analysis is tricky because there are many different ways the code can set \( \text{accept\_sparse} \) to a value other than \text{False}; it is enabled by our novel interprocedural propagation. Figure 4 illustrates the call chain in \text{LogisticRegression} with simplified control flow.

Our analysis either reports (1) a constraint \text{not sp.issparse}(X) at the top of \text{fit/predict}, meaning that if the user passes a sparse matrix the exception is raised, or (2) \text{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 \text{accept\_sparse} to an appropriate value. E.g., in Figure 4 L34, the operator sets \text{accept\_sparse} to ‘\text{csr}’.

Case (1) indicates that the operator’s \text{fit/predict} method does not accept sparse data and this ought to be specified in the docstring. If the analysis reports \text{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 \text{fit/predict} may accept sparse \( X \), as data validation code appears to accept sparse \( X \). If the analysis reports \text{True}, but the docstring states that the method does not accept 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 \text{fit} and \text{predict} methods in all sklearn operators. We detected 2 instances of case (1), one in \text{AffinityPropagation.predict} and one in \text{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 \text{accept\_sparse}, so it defaulted to \text{False}, which triggered the exception when a sparse \( X \) was passed. We reported the issue in \text{AffinityPropagation} to sklearn and suggested the following fix: \text{check\_array}(X, accept\_sparse='\text{csr}')

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 \text{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 \text{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 \text{false positive} if it passes and the hyperparameter configuration is invalid against the JSON schema.
- A trial is a \text{false negative} if it fails and the hyperparameter configuration is valid against the JSON schema.
- A trial is a \text{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 missparse 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 7.5) — 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 (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).

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Time (seconds)</th>
<th>File Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CE</td>
<td>no CE</td>
</tr>
<tr>
<td>AdaBoostClassifier</td>
<td>2,564.86</td>
<td>3,005.27</td>
</tr>
<tr>
<td>AdaBoostRegressor</td>
<td>2,197.82</td>
<td>3,196.64</td>
</tr>
<tr>
<td>BaggingClassifier</td>
<td>2,645.32</td>
<td>3,607.13</td>
</tr>
<tr>
<td>ColumnTransformer</td>
<td>405.58</td>
<td>424.61</td>
</tr>
<tr>
<td>ExtraTreesClassifier</td>
<td>2,828.72</td>
<td>3,969.06</td>
</tr>
<tr>
<td>StandardScaler</td>
<td>4,204.84</td>
<td>38,731.67</td>
</tr>
<tr>
<td>TfidfVectorizer</td>
<td>0.90</td>
<td>1.58</td>
</tr>
<tr>
<td>VotingClassifier</td>
<td>1.02</td>
<td>1.85</td>
</tr>
</tbody>
</table>

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(If, 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 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 [35] 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 [35] 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 techniques 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]. Kazeronian 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 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], [40], 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 [38] 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.

DATA AVAILABILITY

The source code of the analysis and the experiments are available at Zenodo [33].

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


[38] 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
