Principled and practical static analysis for Python: Weakest precondition inference of hyperparameter constraints

Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, Julian Dolby

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

EXTENDED CONFERENCE PAPER

Principled and Practical Static Analysis for Python: Weakest Precondition Inference of Hyperparameter Constraints

Ingkarat Rak-amnouykit*[1] | Ana Milanova[1] | Guillaume Baudart[2] | Martin Hirzel[3] | Julian Dolby[3]

[1]Rensselaer Polytechnic Institute, New York, USA
[2]DI ENS, École normale supérieure, PSL University, CNRS, INRIA, Paris, France
[3]IBM Research, New York, USA

Correspondence
*Ingkarat Rak-amnouykit, 110 8th Street, Troy, NY 12180. Email: rakami@rpi.edu

Application programming interfaces often have correctness constraints that cut across multiple arguments. Violating these constraints causes the underlying code to raise runtime exceptions, but at the interface level, these are usually documented at most informally. This paper presents novel principled static analysis and the first interprocedural weakest-precondition analysis for Python to extract inter-argument constraints. The analysis is mostly static, but to make it tractable for typical Python idioms, it selectively switches to the concrete domain for some cases. This paper focuses on the important case where the interfaces are machine-learning operators and their arguments are hyperparameters, rife with constraints. We extracted hyperparameter constraints for 429 functions and operators from 11 libraries and found real bugs. We used a methodology to obtain ground truth for 181 operators from 8 machine-learning libraries; the analysis achieved high precision and recall for them. Our technique advances static analysis for Python and is a step towards safer and more robust machine learning.

KEYWORDS:
Python, machine learning libraries, interprocedural analysis

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.[1] Incorrect hyperparameter configurations raise exceptions, cause slowdowns, or yield sub-optimal 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,[2] they are also not expressive enough for constraints across multiple hyperparameters, or across hyperparameters and data. Hyperparameter tuning tools, such as auto-sklearn[3] or hyperopt-sklearn,[4] 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.[5] 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.

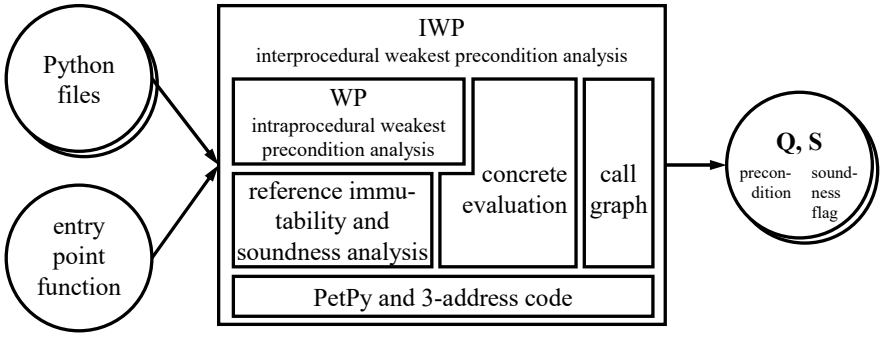

Figure 1 Overview of analysis and its components.

One might be tempted to turn to natural-language documentation for hyperparameter constraints.[6] But even though popular packages like sklearn[1] 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,[1] the most widely-used ML framework today (as of April 2023, the "Used by" count of the sklearn GitHub repository was 494k, ahead of 264k for TensorFlow and 239k for PyTorch). Figure 1 depicts the components of our analysis: 1) PetPy, a minimal Python syntax and 3-address code translation, 2) a call graph analysis to connect functions with call sites; 3) a reference immutability (read/mod) analysis to flag preconditions as sound; 4) WP, an intraprocedural weakest precondition analysis; 5) a concrete evaluation to simplify analysis results; and 6) IWP, an interprocedural weakest precondition analysis. Our preconditions are logic formulas with constraints over 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 static 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, separation logic, compiler theory and static program analysis.

We develop novel, principled, yet practical static analysis for Python. First, we propose PetPy, a minimal Python syntax that abstracts away Python AST constructs. We propose an interpretation of syntactic constructs into *weakest precondition* formulas, which we then turn into hyperparameter constraints. In addition, we propose an interpretation into 3-address code that can serve as input to flow-insensitive static analyses (e.g., reference immutability, taint analysis, and points-to analysis). Our interpretation defines principled rules for the handling of AST constructs; this is in contrast to the majority of work which handles Python AST constructs in an ad-hoc way. Specifically, the analysis interprets certain constructs precisely, while it defaults to a catch-all rule for the remaining, *uninterpreted* constructs; the catch-all rule is unique to our analysis and allows us to reason about unsoundness due to difficult-to-interpret Python construct. We build reference immutability analysis on top of the 3-address code, and a soundness analysis, which help reason about the soundness and "trustworthiness" of the inferred weakest precondition formulas; the 3-address code was necessary to build reference immutability, as the analysis, similarly to many static analyses, is defined over 3-address code. We propose techniques that switch between the analysis domain and the concrete runtime domain to simplify analysis results (weakest precondition formulas). These techniques are particularly suited to Python, which provides rich reflective features that seamlessly integrate static program analysis and dynamic program execution.

We ran our analysis on 429 functions and operators from 11 libraries. Our analysis discovered issues in sklearn, imblearn, TensorFlow, and NumPy, leading to 6 merged pull requests. For the input validation experiments for 181 ML operators from 8 ML libraries (122 sklearn operators plus 59 operators from 7 sklearn-compatible libraries), the analysis achieved 95.6% precision and 75.7% recall, significantly improving over previous work on the problem.

Contributions

This paper makes the following contributions:

- PetPy, a minimal Python syntax for flow-insensitive analysis.
- The first interprocedural weakest precondition analysis for Python defined as an interpretation over PetPy.
- An interpretation into 3-address code, and a reference immutability and soundness analyses built on top of the 3-address code.
- Formula simplification using concrete evaluation.
- Successful application to widely-used machine learning libraries.

We believe that our framework is a step forward towards principled and practical static analysis of Python. In addition, our interface specifications contribute to more reliable and easier to use machine-learning libraries.

Outline

The rest of the paper is organized as follows. Section 2 presents the problem statement and gives an overview of the challenges and proposed analyses. Section 3 presents the PetPy syntax. Section 4 is an overview of the intraprocedural weakest precondition analysis. Section 5 presents our call graph analysis, necessary for the interprocedural weakest precondition analysis. Section 6 details interpretation into 3-address code and the reference immutability and soundness analysis built on top of it. Section 7 puts these building blocks together into the *interprocedural* weakest precondition analysis. Section 8 describes our implementation and the Python features that power the implementation. Sections 9 and 10 describe case study and evaluation; we apply the analyses on widely used ML operators. Finally, Sections 11 discusses related work and Section 12 concludes.

2 | PROBLEM STATEMENT AND OVERVIEW OF TECHNIQUES

This section illustrates our approach for extracting hyperparameter constraints using a weakest-precondition analysis of Python code. As a running example, Figure 2 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.

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 2, the constraint on solver and penalty is also rephrased in the description of penalty: *If 'none' (not supported by the liblinear solver)* (L20).

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.

2.2 | Static Analysis

This section illustrates the main analysis components from Figure 1.

```
from ..utils.multiclass import check_classification_targets
[...]

def _check_solver(solver, penalty, dual):
    if solver not in ['liblinear', 'saga'] and penalty not in ('l2', 'none'):
        raise ValueError(f"{solver} supports only 'l2' or 'none' penalties, got {penalty}.")
    [...]
    return solver


class LogisticRegression(LinearClassifierMixin, SparseCoefMixin, BaseEstimator):
    """
    Logistic Regression (aka logit, MaxEnt) classifier.

    Parameters
    ----------
    penalty : {'l1', 'l2', 'elasticnet', 'none'}, default='l2'
      Used to specify the norm used in the penalization. The 'newton-cg',
      'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is
      only supported by the 'saga' solver. If 'none' (not supported by the
      liblinear solver), no regularization is applied.

    solver : {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'
      Algorithm to use in the optimization problem.
      - 'newton-cg', 'lbfgs', 'sag' and 'saga' handle L2 or no penalty
      - 'liblinear' does not support setting penalty='none'

    l1_ratio : float, default=None
      The Elastic-Net mixing parameter, with 0 <= l1_ratio <= 1. Only
      used if penalty='elasticnet'.
    [...]
    """
    def __init__(self, penalty='l2', solver='lbfgs', l1_ratio=None, [...]):
        self.penalty = penalty
        self.solver = solver
        self.l1_ratio = l1_ratio
        [...]

    def fit(self, X, y, sample_weight=None):
        solver=_check_solver(self.solver, self.penalty, self.dual)

        if self.penalty == 'elasticnet':
            if not isinstance(self.l1_ratio, numbers.Number) or self.l1_ratio < 0 or self.l1_ratio > 1:
                raise ValueError(f"l1_ratio must be between 0 and 1; got {self.l1_ratio}")

        X, y = self._validate_data(X, y, [...])
        check_classification_targets(y)
        self.classes_ = np.unique(y)
        [...]
```

Figure 2 Source excerpt of LogisticRegression operator from https://github.com/scikit-learn/scikit-learn/blob/15a949460/sklearn/linear_model/_logistic.py#L1012.

Intraprocedural Weakest Precondition Analysis

For each operator method, the analysis computes the precondition of each raise exception statement. For instance, in Figure 2, the fit method can raise a ValueError (L44). 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.

Call Graph Construction

Sklearn often externalizes complex checks in dedicated functions. For instance, in Figure 2, 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) (L48).

Reference Immutability 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 Q at L41: 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 L44 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'] ∨ penalty ∈ ['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). 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 7.

Overall, on the example of Figure 2 the analysis yields the following two constraints which match the docstring of the operator constructor.

- penalty = 'elasticnet' ⇒ is_number(l1_ratio) and (0 <= l1_ratio <= 1)
- solver in ['liblinear', 'saga'] or penalty in ['l2', 'none']

3 | PETPY PYTHON SUBSET

Due to the rich features of Python, we focus the analysis on a subset of Python that we call PetPy. Figure 3 presents the syntax. For simplicity, we omit newlines and indentation and use an explicit sequence delimiter.

A program is a sequence of statements. A statement is either nothing (ϵ or pass), a sequence (e.g., x[0] = y; x[1] = z), an assignment (e.g., x = 42), raising an exception (raise), returning a value (return), an if-then-else, a for loop, or a function definition (def). In function definitions, arguments can be positional arguments (e.g., def f(x, y)), an optional variable-length positional argument (e.g., def f(x, *args) where args is a tuple of arguments values), keywords arguments with default values (e.g., def f(x=42, y=0)), and an optional variable-length keyword argument (e.g., def f(**kwargs) where kwargs is a dictionary mapping arguments names to values).

An expressions is either a constant (c, e.g., None or 42), a variable (x), a function call (e.g., f(42)), accessing an attribute (e.g., foo.bar), accessing an element (e.g, foo[0]), safely accessing an element with a default value (e.g., foo.get(0,42) returns 42 if

$$
\begin{aligned}
\mathit{stmt} \;&::= \varepsilon \mid \mathsf{pass} \mid \mathit{stmt}; \mathit{stmt} \mid \mathit{pat} = \mathit{exp} \\
&\mid \mathsf{raise}\ \mathit{exp} \mid \mathsf{return}\ \mathit{exp} \\
&\mid \mathsf{if}\ \mathit{exp} : \mathit{stmt}\ \mathsf{else}\text{:}\ \mathit{stmt} \mid \mathsf{for}\ \mathit{exp}\ \mathsf{in}\ \mathit{exp} : \mathit{stmt} \\
&\mid \mathsf{def}\ f\ (\mathsf{x},)^*\ (\mathsf{*x},)^?\ (\mathsf{x} = \mathit{exp},)^*\ (\mathsf{**x})^?) : \mathit{stmt} \\
&\mid \mathit{Other}\ ((\mathit{stmt})^*) \\
\\
\mathit{exp} \;&::= \mathsf{c} \mid \mathsf{x} \mid \mathit{exp}((\mathit{exp},)^*) \\
&\mid \mathit{exp}.\mathsf{x} \mid \mathit{exp}[\mathit{exp}] \mid \mathit{exp}.\mathsf{get}(\mathit{exp}, \mathit{exp}) \\
&\mid [(\mathit{exp},)^*] \mid \{(\mathit{exp}\text{:}\ \mathit{exp},)^*\} \\
&\mid \mathit{Unop}(\mathit{exp}) \mid \mathit{Binop}(\mathit{exp}, \mathit{exp}) \mid \mathsf{*}\mathit{exp} \mid \mathsf{**}\mathit{exp} \\
&\mid \mathit{Other}\ ((\mathit{exp})^*) \\
\\
\mathit{lhs} \;&::= (\mathit{pat},)^*\ (\mathsf{*}\mathit{pat},)^{!?}\ (\mathit{pat},)^* \\
\mathit{pat} \;&::= \mathsf{x} \mid (\mathit{lhs}) \mid [\mathit{lhs}] \mid \mathit{exp}.\mathsf{x} \mid \mathit{exp}[\mathit{exp}]
\end{aligned}
$$

Figure 3 Syntax of PetPy, a subset of Python. The notation $.^?$ indicates an optional term and $.^*$ indicates a possibly empty sequence. For simplicity, these operators automatically remove trailing commas (e.g., for function arguments) and we use the notation $.^{!?}$ to explicitly leave the trailing comma (e.g., for star assignments).

```
# {(self.penalty=='elasticnet' ⇒ (isinstance(self.l1_ratio,Number) ∧ 0<=self.l1_ratio<=1))
# ∧ (¬(self.penalty=='elasticnet') ⇒ True)
# ≡ self.penalty=='elasticnet' ⇒ (isinstance(self.l1_ratio,Number) ∧ 0<=self.l1_ratio<=1)}
if self.penalty == 'elasticnet':
    # {((¬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)
    # ≡ isinstance(self.l1_ratio,Number) ∧ 0<=self.l1_ratio<=1}
    if not isinstance(self.l1_ratio,Number) or self.l1_ratio<0 or self.l1_ratio>1:
        # {False}
        raise ValueError(f"l1_ratio must be between 0 and 1; got {self.l1_ratio}")
```

Figure 4 Inferring the weakest precondition for one of the exceptions raised by Logistic Regression.

foo[0] does not exist), a list (e.g., [0,1,2]), a dictionary (e.g., {"foo": 42, "bar": 0}), the application of a unary or binary operator (e.g, 2 + 3), unpacking a sequence (e.g., [0, ∗[1,2]] returns [0,1,2]), and unpacking a dictionary ({"foo": 42, ∗∗{"bar": 0}} returns {"foo": 42, "bar": 0}). The left hand-side of an assignment can be used to deconstruct a pattern (e.g., x,y,z = [0,1,2]). A star can be used once in any sequence to capture all remaining elements (e.g., after x,∗y,z = [1,2,3,4] we have y = [2,3]). A pattern is either a variable (x), a tuple, a list, an object attribute (e.g., foo.bar), or an object element (e.g., foo[0]).

We add a new construct *Other* to capture uninterpreted statements and expressions. We thus capture all Python features that are not handled by our analyses. These include while, del, try, and the rest of the Statement nodes specified by the Python AST.

4 | INTRA-PROCEDURAL WEAKEST PRECONDITION ANALYSIS

Our intra-procedural weakest precondition analysis uses essentially standard backwards reasoning[7,8,9] and adapts it for Python. This section describes the core analysis and Section 6 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 or method to prevent the exception. Each step of backward reasoning computes $\mathrm{WP}(\mathit{stmt}, Q_{\mathrm{post}}) \mapsto Q_{\mathrm{pre}}$: given a statement and a postcondition Q_{post}, our analysis returns a precondition Q_{pre} as a valid PetPy expression. WP is defined by induction on the PetPy syntax. PetPy sequences are handled as it is standard in backwards reasoning, while other constructs fall under catch-all rule Other.

- WP(raise *exp*, Q_{post}) handles Raise statements:
  return False

  At a blank Raise statement, an exception is certain, so the precondition for not raising it is $Q_{\text{pre}} = \text{False}$.

- WP($x = exp$, Q_{post}) handles Assignment statements:
  return $Q_{\text{post}}[exp/x]$

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

- WP($exp1.f = exp2$, Q_{post}) and WP($exp1[exp2] = exp3$, Q_{post}) default to the catch-all rule Other (last rule in this list).

  These patterns trigger a modification to a heap variable and we cannot substitute directly into the formula. They default to Other which simply propagates the formula up.

- WP($(lhs) = exp$, Q_{post}) and WP($[lhs] = exp$, Q_{post}) handles tuple and list assignments.

  If *lhs* contains a star pattern, directly or indirectly, then the assignment is handled by Other; for example, x, *y, z = [1,2,3] and x, (y, *z), w = [1,(2,3),4] are handled by Other. As previewed earlier, this means that the postcondition propagates as is. Otherwise, i.e., if there is no star pattern, the assignment recurses down to handling of individual components; for example, x, y[0] = [1,2] recurses to the WP rule for sequence x = [1,2][0]; y[0] = [1,2][1].

- WP(if *exp* : *stmt1* else: *stmt2*, Q_{post}) handles If statements:
  $Q_{\text{pre1}} \leftarrow \text{WP}(stmt1, Q_{\text{post}})$
  $Q_{\text{pre2}} \leftarrow \text{WP}(stmt2, Q_{\text{post}})$
  return $(exp \Rightarrow Q_{\text{pre1}}) \wedge (\neg\, exp \Rightarrow Q_{\text{pre2}})$

  The precondition is standard. If the condition evaluates to True, then the weakest precondition of *stmt1* and Q_{post} must hold, otherwise, the weakest precondition of *stmt2* and Q_{post} must hold.

- WP(for *exp1* in *exp2* : *stmt*, Q_{post}) handles For statements:
  if $Q_{\text{post}} == \text{True}$ **then**
  $Q_{\text{body}} \leftarrow \text{WP}(stmt, \text{True})$
  $Q_{\text{pre}} \leftarrow exp1 \in exp2 \Rightarrow Q_{\text{body}}$
  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 $exp1 \in exp2$. Variables in *exp1* 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 function (fit in our client analysis), we ignore because the clients of weakest precondition analyses do not support such functionality. On the other hand, if postcondition Q_{post} is not True, we simply propagate Q_{post} past the For statement.

- WP(*Other*, Q_{post}) handles Other statements:
  return Q_{post}

  Other statements are Python statements that do not match the syntax of PetPy. The code for Other propagates Q_{post} as-is, which is potentially unsound as the statement may modify locations referenced in Q_{post}. We address this issue in Section 6, where we introduce principled handling of Other statements.

Figure 4 illustrates with the example from Figure 2 L42-L44. 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 $Q_{\text{pre}} = \text{WP}(stmt, Q_{\text{post}})$. Furthermore, Figure 4 indicates formula simplification by showing equivalent formulas, notated as $\{Q_{\text{unsimplified}} \equiv Q_{\text{simplified}}\}$.

5 | 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,[10] and several GitHub repositories, most notably code2flow.[11] 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 and calls on receiver objects, e.g., enc.fit().

The call graph is a parameter to the interprocedural analyses we built in Section 6 and Section 7. We plug in our call graph construction, which we built for the purposes of the interprocedural weakest precondition analysis and consider a minor contribution of the framework; other call graph construction analyses can be used as well.

Basic Algorithm

We propose a new call graph construction analysis that runs in seconds and achieves good accuracy for our purposes. 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. (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:

$$\begin{array}{lll} \text{classTable} & : \text{package:class} & \rightarrow [\text{base1},...,\text{baseN}] \\ \text{functionTable} & : \text{package:class:function} & \rightarrow \text{function code} \end{array}$$

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 2 (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 source code of that function definition. For example, function fit (L39) is represented in functionTable as

'linear_model/_logistic.py:LogisticRegression:fit' → def 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' → def _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 is defined by induction on the PetPy syntax. The main difficulty is that Python has rich syntax and semantics for function calls. Consider the call check_classification_targets(X) in Figure 2 L47, 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). In addition to (1) Name calls, the analysis also matches (2) constructor calls, e.g, LogisticRegression(), (3) calls through self, e.g., self._validate_data(X, y, [...]) (L46), and (4) package-qualified calls, e.g., linear_model._logistic.check_solver(). When matching calls through self, the analysis makes use of classTable to crawl the hierarchy and find functions that are defined in superclasses.

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.[12] 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.

Concrete Evaluation

A 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 for the analysis. 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 out of scope for the analysis?*

We have a simple heuristic 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, such as a NameError exception, then the call remains unresolved. As an example, consider the call np.unique(y) (Figure 2, L48). 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. Distinguishing *external* and *unresolved* calls makes a distinction for what is largely "unachievable" for the analysis (external calls) and what is a potential limitation and presents room for improvement (unresolved). A client analysis may choose different handling for these two categories of calls.

6 | 3-ADDRESS CODE, REFERENCE IMMUTABILITY AND SOUNDNESS ANALYSIS

Side-effects in Python make the weakest precondition analysis from Section 4 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 4 with a *soundness flag*. Clearly, "soundness" is too strong a word given Python's complex semantics and dynamic nature. The notion of soundness that we advance is predicated upon assumptions about call graph correctness and behavior of library calls.

The rest of this section is organized as follows. Section 6.1 describes the soundness flag and propagation of soundness using the *mod* and *read* sets. As mentioned earlier the *mod* set approximates the locations a statement modifies and the *read* set approximates the locations a formula reads. Computation of *mod* and *read* sets uses a known static analysis, *reference immutability*, however extensions are necessary to adapt to the problem of soundness analysis. Section 6.2 outlines standard reference immutability and novel extensions necessary for our purpose. Section 6.3 describes an interpretation of PetPy into 3-address code. This is necessary because reference immutability takes 3-address code as input. It takes a 3-address code statement and creates corresponding constraints for that statement. Translation into 3-address code is non-trivial; our translation can be used for reference immutability analysis as well as a range of other flow analyses such as for example points-to analysis and taint analysis. Finally, Section 6.4 describes fragment reference immutability, yet another extension of the standard analysis, and Section 6.5 outlines the computation of *mod* and *read* sets as a client of reference immutability.

6.1 | 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 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 Sequence, Raise, Assignment, If, and Other statements. PetPy statements that are not defined here (e.g., For) are handled by Other.

- $\text{WP}(\underline{stmt1; stmt2}, Q_{\text{post}}, S_{\text{post}})$ handles Sequences:
  $Q_1, S_1 \leftarrow \text{WP}(stmt2, Q_{\text{post}}, S_{\text{post}})$
  return $\text{WP}(stmt1, Q_1, S_1)$

This is the standard sequence rule in backwards analysis. It propagates the postcondition and combines the soundness flags. If *stmt2* turns the flag to False, then the flag remains False after processing of *stmt1*.

- WP(raise *exp*, Q_{post}, S_{post}) handles Raise statements:
  return (False, True)

  A blank raise makes an exception certain, so the precondition for not raising it is Q_{pre} = False with soundness flag S_{pre} = True.

- WP($x = exp$, Q_{post}, S_{post}) handles Assignment statements:
  $Q_{\text{pre}} \leftarrow Q_{\text{post}}[exp/x]$
  return (Q_{pre}, $S_{\text{post}} \wedge mod(exp) \cap read(Q_{\text{pre}}) == \emptyset$)

  Our treatment follows the principles of separation logic.[13] If the set $mod(exp)$ of locations *modified* by *exp* and the set $read(Q_{\text{pre}})$ 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 $x = exp$, Q_{post} evaluates to true. Otherwise, Q_{pre} is potentially unsound. In other words, if *exp* has side-effects that modify some location referenced by Q_{pre}, then making Q_{pre} true before the execution of the right-hand-side expression *exp* does not necessarily make Q_{post} true after. Section 6.2 describes how to compute *mod* and *read* sets.

- WP(*exp1*.f = *exp2*, Q_{post}, S_{post}) and WP(*exp1*[*exp2*] = *exp3*, Q_{post}, S_{post}) default to Other, as in Section 4.

  These patterns trigger a modification to a heap variable and we cannot substitute into the formula. The formula propagates up, however, the soundness flag will turn to False if the locations modified in the assignment statement and the locations read in the formula intersect.

- WP((*lhs*) = *exp*, Q_{post}, S_{post}) and WP([*lhs*] = *exp*, Q_{post}, S_{post}) handles tuple and list assignments.

  Again, handling is analogous to Section 4, adding the handling of the soundness flag. If *lhs* contains a star pattern, directly or indirectly, the tuple or list assignment is handled by Other. For example, x, *y, z = [1,2,3] and x, (y, *z), w = [1,(2,3),4] are handled by Other, propagating the postcondition as is and possibly turning the flag to False.

  Otherwise, the assignment recurses down to the components, e.g., x, y[0] = [1,2] recurses to the WP rule for sequence x = [1,2][0]; y[0] = [1,2][1].

- WP(if *exp* : *stmt1* else: *stmt2*, Q_{post}, S_{post}) handles If statements:
  $(Q_1, S_1) \leftarrow$ WP(*stmt1*, Q_{post}, S_{post})
  $(Q_2, S_2) \leftarrow$ WP(*stmt2*, Q_{post}, S_{post})
  $Q_{\text{pre}} \leftarrow (\mathsf{E} \Rightarrow Q_1) \wedge (\text{not } exp \Rightarrow Q_2)$
  return (Q_{pre}, $S_1 \wedge S_2 \wedge mod(exp) \cap read(Q_{\text{pre}}) == \emptyset$)

  It is sound if (1) Q_{post} is sound, (2) neither *stmt1* nor *stmt2* contain statements that invalidate the soundness, and (3) *exp* has no effect on Q_1 or Q_2. In practice, *exp* is almost always side-effect free.

- WP(*Other*, Q_{post}, S_{post}) handles all other statements:
  return (Q_{post}, $S_{\text{post}} \wedge mod(other) \cap read(Q_{\text{post}}) == \emptyset$)

  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 (loops are a known thorn in weakest precondition inference), we consider the much easier problem of intersecting the *read* and *mod* sets. If the analysis can show that the weakest precondition formula and the statement do not interfere, then loops and other constructs are handled safely without complex analysis that approximates fixpoint computation.

Figure 5 illustrates this with a constraint in ExtraTreeClassifier that our analysis discovered. It was undocumented and we submitted a pull request which the developers immediately merged. The exception in L9 yields the formula in L5-6. Propagating over the call in L4 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 L3. The call assignment in L2 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.

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

Figure 5 Undocumented constraint in sklearn ExtraTreeClassifier (before the pull request).

6.2 | Reference Immutability

Section 6.2.1 describes the standard reference immutability system ReIm. Section 6.2.2 describes an extension that improves precision when reasoning about modified locations.

6.2.1 | Standard Reference Immutability (ReIm)

The analysis requires information about the set of locations *modified* by a statement *stmt* (actually, a statement sequence) as well as the set of locations *read* by a formula Q. In our treatment statements and formulas 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, however, adaptation to Python requires a significant extension of this work. In this section we outline the ReIm[14] reference immutability system; just as other reference immutability systems such as Javari,[15] ReIm is designed for Java. In the following sections we describe the extensions needed to adapt the analysis to Python and the notion of soundness we use in this work.

ReIm assigns type qualifiers to each *variable*, each *field*, and each *return value* in the program. The two main qualifiers are mutable and readonly:

- mutable: A mutable reference x can be used to mutate the referenced object. This is the implicit and only option in standard languages using the reference model for variables. A mutable reference x indicates that there may be a sequence of assignment statements that leads to a modification to the object *o* that x refers to, or to one of *o*'s components. More precisely, the sequence gives rise to a flow path from x to a y, where y is the receiver of an update: e.g., y.f = z. In all examples below there is a path from x to the corresponding update indicating that the object that x refers to is modified through x:
  - In x.f=1 and in x[0]=1, x is a mutable reference.
  - In y=id(x); y[i]=1, where id is the standard identity function that returns its argument, x is mutable. The flow path in this case is as follows: x → id.param → id.ret → y.
  - In y=x.f; y.g=0, x is mutable. The offending flow path in this case is x.f → y.
- readonly: readonly captures "deep" immutability. A readonly reference x cannot be used to mutate the referenced object nor anything it references, or in other words, there is no sequence of statements that leads to an update of the object x refers to, or to one of its components. All of the following are forbidden:
  - x.f = y
  - x.set(z) where set sets a field of its receiver x. The call creates a flow path x → self, where self is the implicit parameter of set.
  - z = id(x); z.f = w
  - y = x.f; y.g = z

The subtyping relation is mutable <: readonly, meaning that a mutable reference can be assigned to a readonly one, but a readonly reference cannot be assigned to a mutable one. Qualifiers can also be polymorphic, i.e., either mutable or readonly depending on the context.

$$\text{TASSIGN}\quad \frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad q_\mathsf{y} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}$$

$$\text{TWRITE}\quad \frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad q_\mathsf{x} = \mathsf{mutable} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \qquad q_\mathsf{y} <: q_\mathsf{x} \rhd q_\mathsf{f}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}$$

$$\text{TREAD}\quad \frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \qquad q_\mathsf{x} \rhd q_\mathsf{f} <: q_\mathsf{y}}{\Gamma \vdash \mathsf{y} = \mathsf{x.f}}$$

$$\text{TCALL}\quad \frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{m}) = q_\mathsf{p} \to q_\mathsf{ret} \qquad q_\mathsf{y} <: q_\mathsf{x} \rhd q_\mathsf{p} \qquad q_\mathsf{x} \rhd q_\mathsf{ret} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{m(y)}}$$

Figure 6 Typing rules. Function *typeof* retrieves the qualifiers of fields and methods. Γ is a type environment that maps variables to their immutability qualifiers. The $\rhd$ operation at field accesses and at method calls adapts polymorphic fields and polymorphic parameters and return variables, respectively, to the corresponding context of usage.

Figure 6 presents the typing rules following ReIm. They enforce the standard rule that the right-hand side of an implicit or explicit assignment is a subtype of the left-hand side, which disallows flow paths from a readonly variables to an update. The details of the analysis are described by Huang et al.[14] and Milanova.[16]

Standard reference immutability provides type inference as well. The algorithm initializes variables and fields to the full set of qualifiers, then solves the constraints via fixed-point iteration. This entails removing infeasible qualifiers from the sets. For example, the constraint at field write x.f = y requires that x is mutable. Thus, the algorithm removes other qualifiers from the set of x. The key property is that the algorithm computes the "best", i.e., the most precise typing for the program. If a variable x is inferred as readonly, that means no sequence of statements exists that gives rise to a flow path from x to an update statement.

Importantly, the input to reference immutability analysis is a 3-address code representation of the program, as outlined in Figure 6. The major challenge facing principled analysis for Python is to translate complex Python AST constructs (what analysis designers have access to) into 3-address code statements (what many static analyses accept as input), as completely and as accurately as possible.

6.2.2 | self.f Reasoning

Our first extension of standard reference immutability ensures more precise reasoning, which is necessary for weakest precondition analysis. Specifically, our extension treats the implicit parameter self in a special way in order to compute more precise *mod* sets. Let us return to the code example of Figure 5. The modification due to call self._validate_estimator() in L4 is due to a field write statement self._base_estimator_ = ... in the body of _validate_estimator(). We need to narrow down the set of modified locations to *mod*(self._validate_estimator()) = { self.base_estimator_ }. Standard reference immutability would conclude that self is mutable and therefore, *mod*(self._validate_estimator()) = { self.∗ } which implies that all fields of self may be modified by a call to _validate_estimator(). As a result, our soundness analysis would conclude that the set of locations modified by the call statement, namely self.∗ and the set of locations read by the formula, namely self.bootstrap and self.oob_score, intersect, and therefore would set the soundness flag to False.

In order to compute more precise sets we extend the typing rules as shown in Figure 7. The extension is a heuristic that takes advantage of the well-known idiom in object-oriented programming, namely, that fields of objects are written and read exclusively through self. The idiom applies to Python code as well. The analysis captures the semantics of modifications through self, which results in the computation of more precise mod sets. Specifically, the new rules split field writes and instance calls into *self* and *non-self* ones and define rules for each kind. Additionally, each instance method m has an associated *self_mod*(m) set that records the field writes to self in m. Recording happens in rule (TWRITE-SELF) which adds a constraint to include the modified location. At the same time, self is *not made mutable* by the update; mutability propagates through *self_mod*(m) sets: (TCALL-SELF) propagates the sets and (TCALL) ensures that receiver y is mutable when *self_mod*(m) of the callee m is not empty.

The extension preserves the properties of reference immutability: if a non-self reference x is inferred as readonly, then there does not exist a sequence of statements that gives rise to a flow path from x to an update; similarly, if a self reference in method m is readonly, and *self_mod*(m) = ∅, then there does not exist a sequence of statements that gives rise to a path to an update. One can show this by induction on the length of the flow path.

$$\text{TWRITE-SELF}\quad \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \{\mathsf{self.f}\} \subseteq \mathit{self\_mod}(\mathsf{m}') \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd q_{\mathsf{f}}}{\Gamma \vdash \mathsf{self.f} = \mathsf{y} \text{ in method } \mathsf{m}'}$$

$$\text{TCALL-NON-SELF}\quad \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \mathit{typeof}(\mathsf{m}) = q_{\mathsf{self}} \to q_{\mathsf{ret}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd q_{\mathsf{self}} \qquad \mathit{self\_mod}(\mathsf{m}) \neq \emptyset \Rightarrow q_{\mathsf{y}} <: \mathsf{mutable} \qquad q_{\mathsf{x}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{m(y)}}$$

$$\text{TCALL-SELF}\quad \frac{\Gamma(\mathsf{self}_{\mathsf{m}'}) = q_{\mathsf{self}_{\mathsf{m}'}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \mathit{typeof}(\mathsf{m}) = q_{\mathsf{self}}, q_{\mathsf{p}} \to q_{\mathsf{ret}} \qquad q_{\mathsf{self}_{\mathsf{m}'}} <: q_{\mathsf{x}} \rhd q_{\mathsf{self}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd q_{\mathsf{p}} \qquad \mathit{self\_mod}(\mathsf{m}) \subseteq \mathit{self\_mod}(\mathsf{m}') \qquad q_{\mathsf{x}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{m}(\mathsf{self}_{\mathsf{m}'}, \mathsf{y}) \text{ in method } \mathsf{m}'}$$

Figure 7 Typing rules for more precise location reasoning. Rules for non-self TWRITE, all (including self) TASSIGN, and all TREAD remain as in Figure 6.

6.3 | Reference Immutability for Python: From PetPy to 3-address Code

We have adapted the ReIm reference immutability type system[14,16] for Python and inferred qualifiers for each variable and each field. We use our call graph from Section 5 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. This assumption is motivated by our target applications, machine-learning libraries such as sklearn. The vast majority of library calls are to numpy and scipy functions, which follow this behavior.

The problem of adapting reference immutability for Python is challenging due to the fact that standard reference immutability is defined over a 3-address code representation. To the best of our knowledge, translating Python into a 3-address code CFG representation remains an open problem. The principal challenge is the complex semantics of Python. For example, a subscript expression *exp1*[*exp2*] has the following "fixed" interpretation in Java: *exp1* evaluates into a reference to an array object, say a, *exp2* evaluates into an integer i; *exp1*[*exp2*] fetches the i-th element of a. Such an expression translates into 3-address code in a straight-forward manner: tmp = a[i]. In Python, however, one may override the subscript operation and the interpretation of *exp1*[*exp2*] into 3-address code may remain unknown until runtime. Even a basic operation such as subscript becomes non-trivial to handle.

As hinted by the PetPy grammar (Figure 3), our approach divides Python constructs into *interpreted* and *uninterpreted* ones (shown as *Other* in PetPy). Interpreted constructs are ones for which the analysis defines a precise semantics and interprets those constructs faithfully to that semantics. Those constructs have standard interpretation (semantics) into 3-address code in well-studied languages such as Java. Uninterpreted constructs are ones for which the analysis does not define a semantics. The analysis handles those constructs in an approximate way, where handling may introduce both unsoundness and imprecision; our treatment allows clients of the 3-address representation to incorporate reasoning about unsoundness due to uninterpreted constructs.

Our goal is to design an interpretation that handles *flow-insensitive value flow* as accurately as possible, conceding that it is challenging, perhaps impossible, to design a sound interpretation of all constructs into 3-address code. The key idea in our setting is to designate interpreted and uninterpreted nodes; furthermore, we state precisely the interpretation we apply to both interpreted and uninterpreted nodes.

The interpretation function $\mathcal{I}(\mathit{cstr}, \Gamma)$ takes an arbitrary construct *cstr* and a reference environment Γ. The interpretation of a statement (they are not necessarily 3-address code ones in the Python source) does not return a value but adds new *3-address statements* to a global set S. The interpretation of an expression returns a *set of names* that occur in the expression in a certain way; it may modify S as well.

Worklist Algorithm

The analysis uses the standard worklist algorithm:

Reach ← { main }, *Worklist* ← { main } # Entry point of analysis
$S \leftarrow \emptyset$
while *Worklist* $\neq \emptyset$
    $f \leftarrow$ remove function from *Worklist*
    $\Gamma \leftarrow []$
    $\mathcal{I}(f, \Gamma)$

Starting from a designated main function, the analysis interprets the function as specified earlier. At call sites interpretation has effect on the worklist and reachable functions adding new functions as they become reachable. Since interpretation produces 3-address statements as needed by reference immutability, the analysis creates the corresponding constraints for each statement. We **assume** new functions are interpreted in an empty environment; this is a simplifying assumption as it ignores global variables.

6.3.1 | Statement Interpretation

As mentioned earlier the interpretation of a statement adds new statements to a global set of 3-address statements S. The important invariant is that the statements are exactly as required by reference immutability analysis: the self and non-self versions of TASSIGN, TWRITE, TREAD, and TCALL of Figures 6 and 7, where designated components are variable names referring to memory locations. The analysis creates reference immutability constraints as it creates the 3-address statements.

- $\mathcal{I}(\underline{x = exp; stmt}, \Gamma)$ handles a statement sequence with the first statement in the sequence a Variable assignment:
  $R \leftarrow \mathcal{I}(exp, \Gamma)$
  $S \leftarrow S \cup \{ \mathsf{x} = \mathsf{r} \mid \mathsf{r} \in R \}$
  $\mathcal{I}(stmt, \Gamma + \{\mathsf{x}\})$

  The handling augments the environment with the new local variable x, then interprets the remainder of the sequence in the new environment. Note that (1) we assume, but elide from this writeup, that each variable is uniquely identified by its enclosing function, (2) if x already exist in Γ due to an earlier definition in the same function, then the augmentation has no impact. This treatment allows shadowing: if a variable with the same name x is defined in a nested function, augmentation distinguishes it and adds a new variable to the environment and lookup returns the new variable.

  The interpretation of a sequence when the first statement is something other than a variable assignment proceeds as follows: the analysis interprets the first statement in the incoming environment, then it interprets the remainder of the sequence in the same environment.

- $\mathcal{I}(\underline{exp1.\mathsf{f} = exp2}, \Gamma)$ handles Attribute assignment statements:
  $L \leftarrow \mathcal{I}(exp1, \Gamma)$
  $R \leftarrow \mathcal{I}(exp2, \Gamma)$
  $S \leftarrow S \cup \{ \mathsf{l.f} = \mathsf{r} \mid \mathsf{l} \in L, \mathsf{r} \in R \}$

  This is the standard semantics though *exp1* and *exp2* can be arbitrarily complex. For example, consider the interpretation of x.f.g.h = z if a > 0 else w. Jumping ahead a bit (since the interpretation of expressions is detailed in the following subsection), the interpretation of x.f.g returns { n } where n is a fresh variable corresponding to x.f.g. Interpretation of x.f.g has side effects on S: it contributes 3-address statements {n1 = x.f, n = n1.g }. $\mathcal{I}$(z if a > 0 else w) returns the set {z, a, w}; the if expression is uninterpreted, and the function returns all variables mentioned in the expression. The final step contributes the following 3-address-code statements: {n.h = z, n.h = a, n.h = w}.

- $\mathcal{I}(\underline{exp1[exp2] = exp3}, \Gamma)$: As it is customary in flow-insensitive program analysis, subscript assignments *exp1*[*exp2*] = *exp3* are handled analogously to attribute assignments. We interpret the subscript expression e2, as it may contribute statements in S, however, we ignore the index and create l[] = r statements treating the subscript as a special field.

- $\mathcal{I}((lhs) = exp, \Gamma)$ and $\mathcal{I}([lhs] = exp, \Gamma)$: A tuple or list assignment of the form (*lhs*) = *exp* or [*lhs*] = *exp* reduces to *pat* = *exp* when **pat* appears in the *lhs* list (e.g., x,*y,z = [1,2,3] reduces to y = [1,2,3]). It reduces to and triggers interpretation of *pat* = *exp*[i] when it appears as just *pat* in the list (e.g., x,y,z = [1,2,3] reduces to the interpretation of y = [1,2,3][1]). We note that we aim for flow-insensitive 3-address code; the star pattern will match the list as a whole and will in general introduce infeasible value flow.

As a more detailed example of handling of assignments, consider statement sequence l=[1]; ∗l[0],x=[y]; z=l[0][0]. Tuple assignment ∗l[0],x=[y]; invokes the interpretation of l[0]=[y] which gives rise to 3-address statements n1[]=y, l[]=n1 (again, we jump ahead as the handling of expression [y] is detailed in the following subsection). Statement z=l[0][0] gives rise to n2=l[]; z=n2[]. The bottom line is that interpretation captures the flow path from y to z. If z is mutable, mutability is transferred to y.

- $\mathcal{I}(\underline{\text{return } exp}, \Gamma)$ handles Return statements:
  ret ← fresh variable representing the return of the enclosing method
  $R \leftarrow \mathcal{I}(exp, \Gamma)$
  $S \leftarrow S \cup \{ \text{ ret} = \text{r} \mid \text{r} \in R \}$

  The statement is straightforward. The interpretation of the right-hand-side expression is guaranteed to return variables, and we create a simple variable to variable assignment for ret and each one of those variables.

- $\mathcal{I}(\underline{\text{for } exp1 \text{ in } exp2 : stmt}, \Gamma)$ handles For statements:
  $\mathcal{I}(exp1 = exp2[0]; stmt, \Gamma)$

  We turn the interpretation into a sequence. The key part is to capture the binding of *exp1* to the elements of *exp2*. If *exp1* is a variable, say *i*, then the "artificial" statement $exp1 = exp2[0]$ augments the environment and *i* is in scope when analyzing the body of the for statement.

- $\mathcal{I}(\underline{\text{def } fun(param) : stmt}, \Gamma)$ handles function definitions:
  $\mathcal{I}(stmt, \Gamma + \{param\})$

  The interpretation captures inner functions encountered in the function under analysis. The analysis augments the environment with variables from outer scope when analyzing the body of the inner function fun. The outer-scope variables are implicit parameters.

- $\mathcal{I}(\underline{Other(exp1, \ldots, expn)}, \Gamma)$ handles all other statement statements (e.g., raise, if, or uninterpreted statements) :
  $\mathcal{I}(exp1, \Gamma)$
  …
  $\mathcal{I}(expn, \Gamma)$

  The algorithm simply descends into the components of the statement and extracts the corresponding 3-address code. That means that these statements do not "glue" components soundly and precisely. However recursive descent processes all nested assignments and calls and generates 3-address code that captures value flow. We make the **assumption** (based on empirical observation) that the remaining kinds of statements have little impact on flow-insensitive value flow.

6.3.2 | Expression Interpretation

- $\mathcal{I}(\underline{exp1(exp2)}, \Gamma)$ handles Call expressions:
  n ← fresh variable
  $F \leftarrow resolve(exp1(exp2))$ # Call graph
  $A \leftarrow \mathcal{I}(exp2, \Gamma)$
  $S \leftarrow S \cup \{ \text{ n} = \text{f(a)} \mid \text{f} \in F, \text{a} \in A \}$
  if f ∉ *Reach*
  add f to *Reach* and *Worklist*
  return {n}

  We **assume** that the call graph algorithm resolves the call expression. We elide detailed handling of arguments but note that resolution interprets both *exp1* and *exp2* and creates corresponding 3-address statements. Receiver arguments are extracted during the handling of the call. E.g., suppose the call is x.g.h.fun(a1) and it resolves to a function Module.Class.fun in some class Class; then, x.g.h is passed as an argument expression and assigned to self.

- $\mathcal{I}(\underline{exp.\text{f}}, \Gamma)$ handles Attribute/Subscript expressions:
  n ← fresh variable

$V \leftarrow \mathcal{I}(exp, \Gamma)$
$S \leftarrow S \cup \{\ \mathsf{n} = \mathsf{v.f} \mid \mathsf{m} \in M, \mathsf{v} \in V\ \}$
return $\{\mathsf{n}\}$

- $\mathcal{I}(\underline{[exp]}, \Gamma)$ and $\mathcal{I}(\underline{\{exp\}}, \Gamma)$ handle Tuple/List/Set/Dictionary creation expressions:
  $\mathsf{n} \leftarrow$ fresh variable
  $V \leftarrow \mathcal{I}(exp, \Gamma)$
  $S \leftarrow S \cup \{\ \mathsf{n}[] = \mathsf{v} \mid \mathsf{v} \in V\ \}$
  return $\{\mathsf{n}\}$

- $\mathcal{I}(\mathsf{x}, \Gamma)$ handles Variable:
  return $lookup(\mathsf{x}, \Gamma)$

- $\mathcal{I}(\underline{Other\ (exp1, \ldots, expn)}, \Gamma)$ handles uninterpreted expressions:
  return $\bigcup_{1 \le i \le n} \mathcal{I}(expi)$

  It simply returns the union resulting from the interpretation of the component expressions.

6.4 | Fragment Reference Immutability

Our work introduces *fragment reference immutability*. Fragment reference immutability takes a statement and computes readonly or mutable qualifiers for *that single statement*. The following example illustrates and motivates fragment reference immutability:

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

In ReIm's *global* inference, as described in Section 6.2.1, 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 (1) its scope is not the entire program, but a single statement, and (2) at method calls it makes use of parameter, return types, and outer-scope variables computed by ReIm's global inference; these variables present the summary view of the callee. More precisely, it takes as input a statement and interprets it exactly as described in the previous section, creating corresponding constraints. If the statement under analysis contains a method call, the fragment reference immutability makes use of the parameter/return/outer-scope types of the callee function as inferred by global reference immutability. If parameter p is mutable that means there is a sequence of statements that lead to an update of the argument after the call. Thus, fragment reference immutability creates a constraint $q_a <: \mathsf{mutable}$ rather than $q_a <: q_p$. It does not descend into the callee.

As an example, consider the code excerpt below. Interpretation of g inherits the environment of f, including x. The analysis extracts the straight-forward 3-address code statements and global reference immutability creates the corresponding constraints (shown in comment). If we apply fragment reference immutability to the call statement g(), it will deduce mutability of x.

```
1 def f ():
2     x = {}
3     def g():
4         y = x # q_y <: q_x   mutable <: q_y
5         y["foo"] = 42
6     g()
7     return x
```

6.5 | *mod*(S) and *read*(Q)

We now elaborate on the computation of *mod*(S) and *read*(Q) sets.

mod(S) contains x, x.∗, where x≠self, and self.f. They are added to the set as follows:

- If S contains a variable assignment x = ..., then x is placed into *mod*(S). If x is in *mod*(S) this means that statement S may write *stack location* x.
- x.∗ refers to any *heap location* reachable from x. Here x may refer to self as well as a non-self variable. If fragment reference immutability on S infers x mutable, then x.∗ is placed into *mod*(S). This 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. This can happen due to a field write x.f = ... or due to a call m(x) where the parameter is mutable according to global reference immutability.
- self.f refers to a specific field of self. self.f is added to *mod*(S) if self is not mutable in S but self.f ∈ *self_mod*(m), where m is the enclosing method of S. This means that S may write precisely field f of self and no other part of the object. If the statement wrote more deeply on the heap, then self would have been inferred mutable and we would have added self.* to the *mod* set due to the previous rule.

For example, *mod*(_get_n_samples_bootstrap(...)) is { } because method _get_n_samples_bootstrap is read-only, i.e., parameters are inferred by global reference immutability as read-only leaving all expression arguments read-only in the fragment reference immutability.

Recall that Q is valid PetPy expression that can be analyzed. *read*(Q) contains three kinds of elements x, x.∗, and x.f that are added as follows:

- Any variable x that appears in Q is added to *read*(Q). This means that Q reads stack location x.
- If x appears as a component of an argument expression of some call in Q, then x.∗ is added to *read*(Q). This means that Q may pass x into a function and the function may read parts of the object x refers to.
- If x.f appears as a component of Q, then we add x.f to *read*(Q). This means that Q reads precisely field f of x.

For example, *read*(self.bootstrap or not self.oob_score) equals {self, self.bootstrap, self.oob_score}. self is read in the formula, as well as self.bootstrap and self.oob_score. There are no call expressions and therefore no x.∗ elements are added to the set.

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 *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.

7 | 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 5 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 2 (L39). Its preconditions include the preconditions from its own raise (L44), those from callees _check_solver (L40) and self._validate.data (L46), as well as those from other functions transitively reachable in the call graph.

7.1 | 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 4 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. For positional arguments, the analysis performs a straightforward substitution of arguments or default values. 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:

$$\textbf{return} \begin{pmatrix} Q^1_{\text{callee}}[\texttt{arg1/p1}][\texttt{'default'/p2}], \\ \ldots, \\ Q^n_{\text{callee}}[\texttt{arg1/p1}][\texttt{'default'/p2}] \end{pmatrix}$$

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 $(Q^1_{\texttt{callee}}, S^1_{\texttt{callee}})$ we have

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

7.2 | Handling Keyword Arguments

Substitution becomes more complex with keyword arguments (or named arguments), particularly when a variable-length keyword argument is involved. A variable-length keyword argument, usually named `**kwargs` in Python, allows a function to accept multiple keyword arguments as a dictionary. This dictionary 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, and which of the callee's formal arguments use their default values.

At the first contact where the function call contains a variable-length keyword argument, the analysis can only discover what could possibly be in there by inspecting the callee's function arguments. To handle this, the analysis creates a dictionary placeholder to keep track of possible keyword arguments and their default values, and integrates the placeholder into the current precondition formula. The placeholder in the precondition indicates that the value of the associated variable-length keyword argument could not be determined yet as it can potentially be specified in a future function call of a part of the call graph that the analysis has not reached yet. At such a function call, the placeholder in the precondition is then resolved (see details in Section 8.2).

7.3 | Concrete Evaluation

The handling of keyword arguments 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:

- *constr*(C1,C2)[target/value] handles substitution in an arbitrary construct *constr*:
  $Q' \leftarrow$ *constr*(C1[target/value], C2[target/value])
  $Q'' \leftarrow \text{eval}(Q')$
  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'. Simplify(Q'') performs standard simplification, most notably of implication formulas: False $\Rightarrow$ Q becomes True and True $\Rightarrow$ Q becomes Q.

```
# {¬ isinstance(accept_sparse,str) ⇒ ¬ accept_sparse is False}
def _ensure_sparse_format(spmatrix, accept_sparse, dtype, copy, force_all_finite, accept_large_sparse):
    if isinstance(accept_sparse, str):
        accept_sparse = [accept_sparse]
    if accept_sparse is False:
        raise TypeError('A sparse matrix was passed, but dense data is required. Use X.toarray() to convert to a
            dense numpy array.')

# {¬ isinstance(accept_sparse,str) ⇒ ¬ accept_sparse is False}
def check_array(array, accept_sparse=False, *, accept_large_sparse=True, dtype="numeric",
                order=None, copy=False, force_all_finite=True, ensure_2d=True, allow_nd=False,
                ensure_min_samples=1, ensure_min_features=1, estimator=None):
    array = _ensure_sparse_format(array, accept_sparse=accept_sparse, dtype=dtype, copy=copy,
                                  force_all_finite=force_all_finite, accept_large_sparse=accept_large_sparse)

class BaseEstimator:
    # {¬ isinstance(check_params.get('accept_sparse',False),str)
    # ⇒ ¬ check_params.get('accept_sparse',False) is False}
    def _validate_data(self, X, y='no_validation', reset=True, validate_separately=False, **check_params):
        X = check_array(X, **check_params)

class LogisticRegression(LinearClassifierMixin, SparseCoefMixin, BaseEstimator):
    # {True}
    def fit(self, X, y, sample_weight=None):
        X, y = self._validate_data(X, y, accept_sparse='csr', dtype=_dtype, order="C",
                                   accept_large_sparse=solver!='liblinear')
```

Figure 8 Modified excerpt with simplified path from sklearn LogisticRegression fit function to one specific exception.

8 | IMPLEMENTATION

This section details the implementation of various parts of the analysis.

8.1 | Python AST

As mentioned earlier, the analysis analyzes source code over its abstract syntax tree, making use of the python AST module. There are 2 main usages of AST nodes; to process code for call graph construction (see Section 5) and to store precondition formula during the weakest precondition analysis (see Section 4). First, the analysis crawls through Python files and collects necessary information such as function definition as a mapping from function names to their corresponding AST nodes. This information is then used to construct a call graph starting from a selected target function. During the weakest precondition analysis, formulas of preconditions from each raise exception are stored in AST format because they are easy to process, modify, and compare as they travel up the call graph until reaching the target function. Formulas in AST format are also a good fit for simplification and concrete evaluation, as we can use the *unparse* functionality of the AST library and construct Python code for concrete evaluation.

8.2 | Variable-Length Keyword Arguments

This section provides the implementation detail of handling of variable-length keyword arguments in Section 7.2. To handle this feature, the analyzer uses the built-in get method of Python dictionaries to keep track of values of the keywords arguments. At the entry of a function func with keyword arguments, the analysis creates kwName, a placeholder for a dictionary that will eventually be replaced with an actual dictionary containing the value of keywords arguments. For a keyword argument kw with a default value

df, the analyzer substitutes occurrences of kw in formulas $Q_{\texttt{func}}$ (formulas reaching the entry of func) with kwName.get('kw',df). A parameters that does not occur in the dictionary thus becomes the default value (in Python d.get(a, b) returns b if a is not in d).

Later in the call graph, we arrive at a function with a function call that does not pass actual keyword arguments but passes a dictionary, e.g., **kwargs. Here we substitute kwName with kwargs in all formulas. More formally, we have Q[kwargs/kwName] and kwName.get('kw',df) becomes kwargs.get('kw',df) in Q. Eventually, we arrive at a call that passes actual keyword arguments, typically a subset of all keyword arguments in kwName, e.g., {'kw1':v1, 'kw2':v2}. At this point we can replace the placeholder with this dictionary. Every instance of {...}.get('kw',df) is thus resolved. If a keyword kw *is not* in the actual dictionary {...}, the formula resolves to kw's default value df, the second argument of get. If kw *is* in the actual dictionary {...}, the formula resolves to the corresponding value.

Figure 8 illustrates an example from sklearn LogisticRegression. 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. The values of keyword arguments is not known at the call to check_array (L19); they are only set at the call to validate_data (L18) and passed through in L19 via argument check_params.

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 (L9) contains a call to _ensure_sparse_format. The call does not involve variable-length keyword arguments so the analysis performs a straightforward substitution. The precondition result is the same (L8).

Next, in function _validate_data, a call to check_array involves **check_params (L19). At this point we are not able to determine which keyword arguments are in **check_params. The analysis uses the dictionary name, here check_params, as a placeholder and substitutes all keyword arguments in the condition Q with a safe access using the default value

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 L16–17.

Finally, in fit, the call self._validate_data(..) (L24) 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 from this call path.

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. To mitigate this issue, when the actual keyword argument is *not* a constant, the analysis unparses 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.

8.3 | JSON Schema

After extracting weakest precondition constraints, the analysis can further encode constraints into JSON Schema[17], a widely-supported and widely-adopted schema language. For instance, JSON Schema is the foundation of the Open API language for specifying REST APIs[18]. 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[19].

The JSON Schema translator encodes extracted constraints, stored as Python AST, into JSON Schema format. Unlike Python 3 types, JSON Schema is expressive enough for complex formulas. The encoder removes pythonic expressions, rewrites logical implication, and performs boolean simplification. It also supports min and max functions, and comparison of variables and values.

The extracted constraints capture the relationship between arguments and data, and are particularly useful for argument validation. In a situation where some arguments need to be set beforehand, they can be validated against weakest precondition

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

Figure 9 JSON Schemas of the two constraints extracted from the code of Figure 2

constraints to ensure that exceptions will not be raised. This allows for early detection of errors and avoids encountering them during the runtime.

Recall the excerpt of the sklearn logistic regression operator in Figure 2, the analysis extracts the following two constraints.

- penalty = 'elasticnet' ⇒ is_number(l1_ratio) and (0 <= l1_ratio <= 1)
- solver in ['liblinear', 'saga'] or penalty in ['l2', 'none']

The first constraint is translated into the JSON schema shown in Figure 9 (L1-7). 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". L9-14 shows the JSON schema of the second constraint.

9 | CASE STUDIES

The interprocedural weakest preconditions analysis is effective at finding real issues. We ran the analysis on 11 libraries and tracked discrepancies between documentation and preconditions we inferred from code. We reported a total of 8 issues to developers: 3 issues to sklearn, 1 issue to imblearn, 3 issues to TensorFlow, and 1 issue to NumPy. All issues from sklearn and imblearn were fixed and merged into respective libraries. All 3 TensorFlow issues were acknowledged by developers and we created 1 approved PR addressing one of the issues. For NumPy, we created 1 issue that led to a PR and a change in both the code and the documentation. In addition, we have contributed PRs to the Lale Auto-ML library,[20] specifically improving the JSON schema constraints of 72 of sklearn's operators.

The 11 libraries can be categorized into two groups: (1) Sklearn-compatible libraries and (2) Other libraries. The first group consists of sklearn (122 operators) and 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. These 7 sklearn-compatible libraries follow sklearn convention and provide (operator class, fit) targets. 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. We detail our studies on this group in Section 9.1.

The second group consists of 3 libraries: TensorFlow (200 functions), NumPy (5 functions), and Pandas (43 functions). They are independent libraries that do not follow sklearn convention. For these libraries, we look for classes and functions that have both documentation and potential preconditions. We detail our case studies in Section 9.2.

9.1 | Sklearn and Sklearn-Compatible Libraries

We have reported two hyperparameter issues, one in sklearn and one in imblearn, and discovered inconsistencies in sklearn's sparsity checks. The sklearn issue is an undocumented constraint between hyperparameter bootstrap and oob_score that we previously discussed where an exception is raised if bootstrap==False and oob_score==True. This constraint affects ExtraTreesClassifier and 5 other operators. We created a PR updating the docstrings of oob_score to indicate that it depends on bootstrap. For imblearn, we reported a case in InstanceHardnessThreshold where the estimator parameter does not support a string value contradicting the documentation. The developers issued a PR that updated the documentation to match the precondition.

Moreover, having noticed inconsistencies in sklearn's sparsity checks, we ran an additional experiment. Operators in sklearn run data validation code as illustrated in Figure 8. As shown, validation code checks for sparsity of X with default is accept_sparse =False. We isolated the following exception in 'sklearn/utils/validation.py:None:_ensure_sparse_format' (L6 in Figure 8):

```
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 8 illustrates the call chain in LogisticRegression with simplified control flow.

Our analysis reports either (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 8 L22, 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 8 L24, 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.

9.2 | Additional Libraries

This group consists of independent libraries that do not follow sklearn convention: TensorFlow, NumPy, and Pandas. For these libraries, the entry point of the analysis is a (class, function) tuple. We look for functions that are documented and compare them with preconditions inferred by our IWP analysis.

TensorFlow is a popular machine-learning library. We focused on the neural net module as it is one of the commonly used features. For each function, we compared extracted constraints from the analysis with its documentation. Documented constraints can be specified either in descriptions of function's arguments, or in a separate section detailing possible raise exceptions. For example, in tf.nn.fractional_max_pool, there are 4 exceptions checking if the argument pooling_ratio has correct properties such as it needs to be an integer or a list of integer of length 1, 2, or 4. The argument description of pooling_ratio explains its requirements but does not explicitly state that exceptions will be raised for misconfigured values. On the other hand, a ValueError exception is raised if the argument Seed is not specified and TensorFlow operation is set to be deterministic. This constraint is

clearly stated in the Raises section along with the type of Raise statement that will be thrown. In both cases, we consider that the preconditions, if found, are properly documented. Overall, we discovered 4 functions where the analysis found undocumented constraints. For example, the tf.nn.max_pool function raises an exception if the explicit padding is used with an input tensor of rank 5. We reported 3 issues that were acknowledged by the developers, and created 1 PR to improve the documentation. It was approved and merged into the main branch.

NumPy is a library for working with multi-dimensional arrays. The analysis ran on Python parts of NumPy which contain some exceptions throughout the code. The notable precondition is an undocumented constraint in the memory-map class numpy.memmap. The ValueError is raised if the argument Mode is w+ and the argument Shape is not specified. This constraint is not in the documentation and we reported it to the developers. The PR was created and merged within a week. The changes are both in the code to make the exception message more meaningful, and in the documentation to specify this constraint.

Pandas is a popular data manipulation library. We focused on DataFrame which is the primary API of Pandas. It is a two-dimensional data structure with labeled axes that supports public functions for data manipulation. We analyzed 43 functions in the DataFrame class and did not uncover any discrepancies between documentation and inferred preconditions. A few preconditions involved multiple function arguments. For example, in the from_dict function, there is a constraint between columns and orient arguments inferred from an exception that is raised when columns is set (not None) and orient is specified to certain values.

10 | EVALUATION

This section experimentally evaluates the performance of our analysis using three existing libraries: sklearn and 2 sklearn-compatible libraries, XGBoost and LightGBM. The evaluation focuses on these libraries because 1) the code is almost entirely written in Python and 2) they provide high-quality documentation that can be checked against the generated constraints. Our evaluation considers three research questions:

RQ1 Is the IWP analysis effective for schema validation for ML operators and how does it compare to existing solutions?

RQ2 How well does the soundness analysis work?

RQ3 What is the impact of concrete evaluation and does the IWP analysis scale?

To answer RQ1, we designed a fuzzing mechanism that can sample random configurations for an operator. We then convert the constraints computed by the analysis to a JSON schema[17] and test if a JSON schema checker is able to stop invalid configurations while allowing valid ones. Our analysis achieved 95.6% precision and 75.7% recall and outperforms existing approaches.

To answer RQ2, 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 RQ3, we report on the impact of the concrete evaluation simplifier from Section 7.3 and on analysis running times. The analysis runs in under 11 minutes per operator for all operators, 3 minutes on average.

10.1 | RQ1: Is the IWP 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 JSON 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.[20] 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 then translate weakest precondition constraints from the analysis into JSON Schema 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.

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 sum up trial results from all operators to calculate precision and recall. On these 101 operators, the analysis achieved **95.6% precision** and 75.7% recall leading to an F_1-score of 84.5%. 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. (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 are treated as True. As a result, in our experiments, schemas with inexpressible clauses are nearly always valid, i.e., accept everything. At first, 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 Section 9.1. Unfortunately, the JSON schema for that exception always evaluated to True resulting in thousands of false negatives. With special handling of the schema for that exception, the recall score of those operators increases by about 20 points. Note that 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 10.3) — 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[20] and automatically documentation-extracted schemas.[6] The hand-written constraints are crafted by careful manual examination of the documentation. We use the same experimental methodology to compare IWP to these solutions.

Figure 10 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 10 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 97.8% and 77.3% respectively, resulting in F_1-score of 86.3 as shown in Figure 10. 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 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

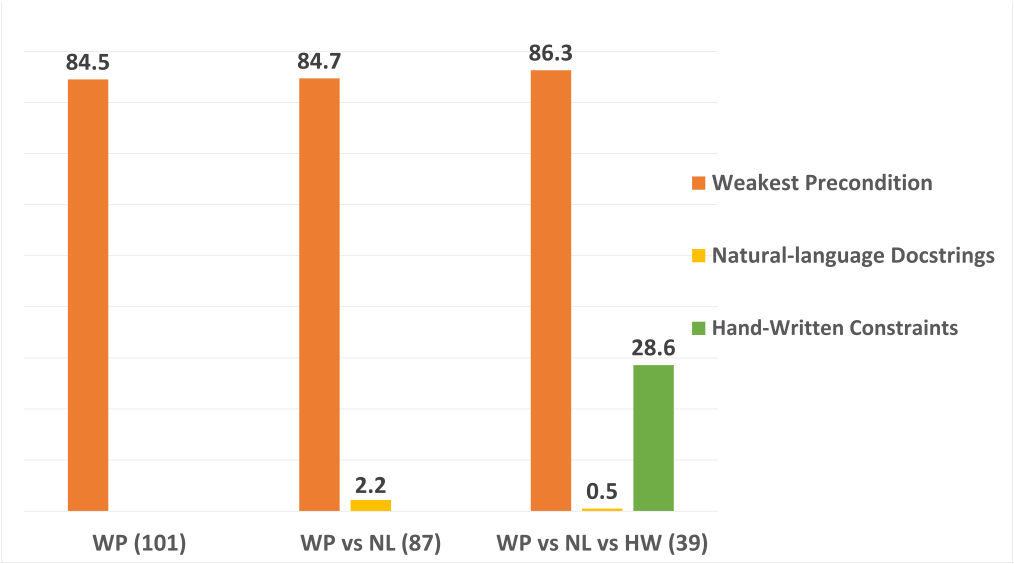

Figure 10 Average F_1-score of 3 groups; the group with all 101 operators (only IWP schemas), the subgroup of 87 operators (IWP and NL Docstrings schemas), and the subgroup of 39 operators (IWP, NL, and Hand-Written schemas).

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 10 shows that our weakest precondition analysis clearly outperforms the technique from Baudart et al.[6]

10.2 | RQ2: Does Soundness Analysis Work?

We ran the IWP analysis on all 181 operators starting at the target (operator,fit). Recall that this turns all 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 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 10.3) — an unsound precondition may pass validation while the exception still fires at runtime and this never happened in our experiments.

Table 1 Running times and file sizes of some operators when applying the concrete evaluation (CE) and the pruning heuristic (P). The files store AST formulas of weakest preconditions in .pkl format. T/O (timeout) means the run time limit of 5 hours was exceeded.

	Time (seconds)				File Size (kB)			
	Plain	CE	P	CE and P	Plain	CE	P	CE and P
LabelEncoder	6s	4s	7s	5s	3	3	3	3
LabelBinarizer	3m18s	4m01s	54s	1m45s	46	44	46	44
SkewedChi2Sampler	41m40s	34m31s	56s	4m00s	173,584	9,187	2,33	921
LassoLarsIC	50m33s	36m46s	1m11s	4m29s	189,96	27,59	2,169	2,687
PassiveAggressiveClassifier	2h15m57s	48m06s	1m28s	4m31s	341,516	6,639	3,8	701
VarianceThreshold	42m14s	38m51s	58s	4m34s	224,16	10,875	2,599	139
SelectKBest	36m06s	37m13s	1m15s	4m36s	214,909	33,970	2,436	2,789
RandomForestClassifier	1h08m23s	1h04m53s	1m20s	4m45s	347,579	131,197	3,121	3,933
FeatureAgglomeration	1h16m31s	57m36s	1m18s	5m10s	496,860	29,178	4,931	2,308
SparseRandomProjection	40m19s	40m05s	57s	5m20s	198,924	10,947	2,324	949
HuberRegressor	1h12m07s	41m32s	1m10s	5m26s	368,834	12,249	3,20	907
LabelSpreading	T/O	1h39m22s	1m46s	5m55s	T/O	104,181	3,778	3,318
IsolationForest	3h14m23s	47m47s	2m04s	5m55s	404,149	11,182	3,903	1,155
SparsePCA	T/O	T/O	2m22s	8m32s	T/O	T/O	5,696	9,613
RidgeClassifier	T/O	T/O	6m30s	14m08s	T/O	T/O	17,142	10,722

10.3 | RQ3: Does the Analysis Scale?

Our analysis, like all analyses in this space, suffers from path explosion. This applies to exceptions that 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 very quickly due to the two techniques (1) concrete evaluation and (2) pruning heuristic.

Concrete evaluation is important for scalability. As discussed in Section 7.3, part of a precondition can sometimes be evaluated during the analysis, mostly due to substitution of default values of formal arguments at a function call. The analysis attempts to perform concrete evaluation after every substitution which maintains the size of the current precondition to be as compact as possible. Nevertheless, preconditions of deeply-nested exceptions can still suffer from path explosion.

Another strength of concrete evaluation is the ability to identify trivial preconditions that are True at the top of the function under analysis. In sklearn, each operator calls an input validation function _validate_data which gives rise to multiple call paths to the check_array function under different conditions; different operators instantiate inputs to check_array differently. Figure 8 shows a modified source excerpt of one call path. An exception can have multiple preconditions from different call paths, and some of them could be True. A weakest precondition True means that its corresponding exception of this specific call path will never be reached. For example in PCA, without concrete evaluation, one precondition is

```
{'no_validation'Is None => NOT({'dtype':[np.float64,np.float32],'ensure_2d':True,'copy':self.copy}.get('
force_all_finite',True)NotIn [True,False,'allow-nan'])}.
```

It is from a call path of an If statement "if y is None:" but PCA does not take y so it uses a default argument value of 'no_validation' in the _validate_data function. Unevaluated preconditions, even when True, take time to ascertain. Concrete evaluation evaluates this to True, reducing number of weakest preconditions to consider.

Pruning. To improve scalability after concrete evaluation, we apply a pruning heuristic that removes formulas that are excessively large, specifically ones that have more than 200 implications at any point during the analysis. Such formulas are uninterpretable and not useful for our client analysis. Pruning is unsound because some preconditions are removed. However, since pruned preconditions are large and not useful for us, it does not affect the experiment. Still, the pruning limit needs to be carefully chosen. Aggressive pruning could improve the running time of the analysis at a cost of potentially removing useful weakest preconditions.

For example, form the hyperparameter experiment, setting the pruning implication number below 24 would remove a useful precondition from PCA. This precondition involves hyperparameter n_components and svd_solver and correctly detect 247 invalid configurations (True Positive case) of PCA's dense dataset. We choose the pruning number of 200 which keeps the running time reasonable while removing excessively large preconditions.

As an ablation study, we measure the running time of the analysis and the size of the result file when applying the two techniques. 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 15 representative sklearn operators picked to uniformly cover the full spectrum of running time of the normal runs. Similar to the previous experiment in the evaluation section, the analysis takes a fit method as an entry point for all operators. The Plain run evidently presents the effect of path explosion. The running time and the file size can become extremely large and exceed the time limit for some operators. When applying only the concrete evaluation, most operators finish under an hour and produce a result file under the size of 30MB. The concrete evaluation actively simplifies preconditions, and the result perfectly captures everything from the analysis; however, it also contains complex weakest preconditions that are too large and uninterpretable. The analysis wastes a considerable amount of time on these preconditions that eventually will not be useful for our client analysis. On the other hand, when applying only the pruning heuristic, the analysis finishes in about a minute, which is much faster than other methods. However, the resulting weakest preconditions are incomplete. Without the concrete evaluation, preconditions rapidly become larger and get pruned by the heuristic too early in the analysis, resulting in singificantly fewer weakest preconditions. Lastly, when combining both the concrete evaluation and the pruning heuristic, the analysis completes in **under 15 minutes on all operators with an average of about 5 minutes**.

We run the experiments on a Windows machine with Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz.

11 | RELATED WORK

Our non-archival preliminary work 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.[21] Next, we built upon and significantly extended this early work 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[22]. This paper further improves the core analysis to increase precision and recall of the experiment. We create PetPy, a subset of Python that allows principled formalization of our analyses. Based on PetPy, we formalize weakest precondition and reference immutability for Python, as well as detail translation from Abstract Syntax into 3-address code for the purposes of flow-insensitive analysis. In addition, we add significant details about our Python implementation and introduce case studies with the application of our framework to 3 new libraries, in addition to the sklearn-family from the previous paper.

While backward reasoning and, more generally, verification condition generation have a long history, with papers by Flanagan et al.[23] and by Branett et al.[9] 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.[24,25] Kazerounian et al.[25] 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.[26] 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,[27] as that evaluates SMT formulas to find inputs that improve coverage. PyExZ3[28] and PyCT[29] 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[30] explores static analysis of machine-learning libraries and outlines challenges to traditional static analysis techniques and Monat et al.[31] present type analysis via abstract interpretation and[32] construct multilanguage analysis for Python with C extensions. Similarly to our work, this work formalizes a subset of Python and designs precise, flow-sensitive and context-sensitive interpretation of this subset. In contrast, our work aims to provide interpretation into 3-address code, which can be used by *flow-insensitive* analyses, e.g., reference immutability and pointer analysis. Monat et al.'s analysis is significantly more precise, while ours is more scalable. In addition, we focus on the specific problem of extracting hyperparameter constraints and develop a practical analysis that extracts logical formulas.

There is a body of work on type inference for Python, including by Maia et al.,[33] Xu et al.,[34] and Hassan et al.[35] Recent work explores machine-learning-based type inference, including work by Allamanis et al.[36] and by Pradel et al.[37] Our work focuses on inference of deeper semantic properties such as hyperparameter and data constraints. iComment for C[38] and jDoctor for Java[39] 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.[40] Habib et al. check data flowing through ML pipelines using JSON subschema checks.[41] 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[3] and hyperopt-sklearn[4] 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.

12 | CONCLUSIONS

This paper presents an interprocedural static analysis for extracting weakest preconditions from Python. Python's dynamic features lead to imprecisions in the analysis that cause path explosion. Therefore, while the core analysis operates over a standard abstract domain of formulas, we extend it with concrete evaluation to make it tractable. The concrete evaluation uses the Python interpreter, thus enlisting one dynamic language feature to help tame other dynamic language features. We add reasoning about soundness using reference immutability, following the principles of separation logic. We have successfully applied the analysis on 181 operators across 8 ML libraries and 248 functions across 3 popular libraries. The analysis achieved high precision and recall in the input validation experiment.

DATA AVAILABILITY

The source code of the analysis and the experiments are available at Github (https://github.com/Ingkarat/IWP).

References

1. Buitinck L, Louppe G, Blondel M, et al. API Design for Machine Learning Software: Experiences from the scikit-learn Project. https://arxiv.org/abs/1309.0238; 2013.

2. Rak-amnouykit I, McCrevan D, Milanova A, Hirzel M, Dolby J. Python 3 Types in the Wild: A Tale of Two Type Systems. Dynamic Languages Symposium (DLS); 2020: 57–70. https://doi.org/10.1145/3426422.3426981.

3. Feurer M, Klein A, Eggensperger K, Springenberg J, Blum M, Hutter F. Efficient and Robust Automated Machine Learning. Conference on Neural Information Processing Systems (NIPS); 2015: 2962–2970. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.

4. Komer B, Bergstra J, Eliasmith C. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. Python in Science Conference (SciPy); 2014: 32–37. http://conference.scipy.org/proceedings/scipy2014/komer.html.

5. Bavishi R, Lemieux C, Fox R, Sen K, Stoica I. AutoPandas: Neural-Backed Generators for Program Synthesis. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); 2019. https://doi.org/10.1145/3360594.

6. Baudart G, Kirchner P, Hirzel M, Kate K. Mining Documentation to Extract Hyperparameter Schemas. ICML Workshop on Automated Machine Learning (AutoML@ICML); 2020. https://arxiv.org/abs/2006.16984.

7. Hoare CAR. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)* 1969; 12(10): 576–580. https://doi.org/10.1145/363235.363259.

8. Leino KRM. Efficient weakest preconditions. *Information Processing Letters* 2005; 93(6): 281–288. https://doi.org/10.1016/j.ipl.2004.10.015.

9. Barnett M, Chang BE, DeLine R, Jacobs B, Leino KRM. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Symposium on Formal Methods for Components and Objects (FMCO); 2005: 364–387. https://doi.org/10.1007/11804192_17.

10. Salis V, Sotiropoulos T, Louridas P, Spinellis D, Mitropoulos D. PyCG: Practical Call Graph Generation in Python. International Conference on Software Engineering (ICSE); 2021: 1646–1657. https://arxiv.org/abs/2103.00587.

11. Rogowski S. code2flow. https://github.com/scottrogowski/code2flow; 2021.

12. Behnel S, Bradshaw R, Citro C, Dalcín L, Seljebotn DS, Smith K. Cython: The Best of Both Worlds. *Computing in Science and Engineering (CISE)* 2011; 13(2): 31–39. https://doi.org/10.1109/MCSE.2010.118.

13. O'Hearn PW, Reynolds JC, Yang H. Local Reasoning about Programs that Alter Data Structures. Workshop on Computer Science Logic (CSL); 2001: 1–19. https://doi.org/10.1007/3-540-44802-0_1.

14. Huang W, Milanova A, Dietl W, Ernst MD. ReIm & ReImInfer: checking and inference of reference immutability and method purity. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); 2012: 879–896. https://doi.org/10.1145/2398857.2384680.

15. Tschantz MS, Ernst MD. Javari: adding reference immutability to Java. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); 2005: 211–230. https://doi.org/10.1145/1094811.1094828.

16. Milanova A. Definite Reference Mutability. European Conference for Object-Oriented Programming (ECOOP); 2018: 25:1–25:30. http://drops.dagstuhl.de/opus/volltexte/2018/9230.

17. Pezoa F, Reutter JL, Suarez F, Ugarte M, Vrgoč D. Foundations of JSON Schema. International Conference on World Wide Web (WWW); 2016: 263–273. https://doi.org/10.1145/2872427.2883029.

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

19. Baudart G, Hirzel M, Kate K, Ram P, Shinnar A, Tsay J. Pipeline Combinators for Gradual AutoML. Advances in Neural Information Processing Systems (NeurIPS); 2021. https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf.

20. Baudart G, Hirzel M, Kate K, Ram P, Shinnar A. Lale: Consistent Automated Machine Learning. KDD Workshop on Automation in Machine Learning (AutoML@KDD); 2020. https://arxiv.org/abs/2007.01977.

21. Rak-amnouykit I, Milanova A, Baudart G, Hirzel M, Dolby J. Extracting hyperparameter constraints from code. ICLR Workshop on Security and Safety in Machine Learning Systems (SecML@ICLR); 2021. https://aisecure-workshop.github.io/aml-iclr2021/papers/18.pdf.

22. Rak-amnouykit I, Milanova A, Baudart G, Hirzel M, Dolby J. The Raise of Machine Learning Hyperparameter Constraints in Python Code. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis; 2022; New York, NY, USA: 580–592. https://doi.org/10.1145/3533767.3534400

23. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R. Extended Static Checking for Java. Conference on Programming Language Design and Implementation (PLDI); 2002: 234–245. https://doi.org/10.1145/543552.512558.

24. Chlipala A. Ur: statically-typed metaprogramming with type-level record computation. Conference on Programming Language Design and Implementation (PLDI); 2010: 122–133. https://doi.org/10.1145/1809028.1806612.

25. Kazerounian M, Guria SN, Vazou N, Foster JS, Horn DV. Type-Level Computations for Ruby Libraries. https://doi.org/10.1145/3314221.3314630; 2019.

26. Hirzel M, Dincklage DV, Diwan A, Hind M. Fast Online Pointer Analysis. *Transactions on Programming Languages and Systems (TOPLAS)* 2007; 29(2). https://doi.org/10.1145/1216374.1216379.

27. Majumdar R, Sen K. Hybrid Concolic Testing. International Conference on Software Engineering: Companion (ICSE-C); 2007. https://doi.org/10.1109/ICSE.2007.41.

28. Ball T, Daniel J. Deconstructing Dynamic Symbolic Execution. Dependable Software Systems Engineering; 2015: 26–41.

29. Chen Y, Tsai W, Wu W, Yen D, Yu F. PyCT: A Python Concolic Tester. Asian Symposium on Programming Languages and Systems (APLAS); 2021: 38–46. https://doi.org/10.1007/978-3-030-89051-3_3.

30. Dolby J, Shinnar A, Allain A, Reinen J. Ariadne: Analysis for Machine Learning Programs. Workshop on Machine Learning and Programming Languages (MAPL); 2018: 1–10. http://doi.acm.org/10.1145/3211346.3211349.

31. Monat R, Ouadjaout A, Miné A. Static Type Analysis by Abstract Interpretation of Python Programs. European Conference on Object-Oriented Programming (ECOOP); 2020: 17:1–17:29. https://drops.dagstuhl.de/opus/volltexte/2020/13208.

32. Monat R, Ouadjaout A, Miné A. A Multilanguage Static Analysis of Python Programs with Native C Extensions. Static Analysis; 2021; Cham: 323–345. https://doi.org/10.1007/978-3-030-88806-0_16.

33. Maia E, Moreira N, Reis R. A Static Type Inference for Python. Workshop on Dynamic Languages and Applications (DYLA); 2011. http://scg.unibe.ch/download/dyla/2011/dyla11_submission_3.pdf.

34. Xu Z, Zhang X, Chen L, Pei K, Xu B. Python Probabilistic Type Inference with Natural Language Support. Symposium on the Foundations of Software Engineering (FSE); 2016: 607–618. https://doi.org/10.1145/2950290.2950343.

35. Hassan M, Urban C, Eilers M, Müller P. MaxSMT-Based Type Inference for Python 3. Conference on Computer Aided Verification (CAV); 2018: 12–19. https://doi.org/10.1007/978-3-319-96142-2_2.

36. Allamanis M, Barr ET, Ducousso S, Gao Z. Typilus: Neural Type Hints. Conference on Programming Language Design and Implementation (PLDI); 2020: 91–105. https://doi.org/10.1145/3385412.3385997.

37. Pradel M, Gousios G, Liu J, Chandra S. TypeWriter: Neural Type Prediction with Search-Based Validation. Symposium on the Foundations of Software Engineering (FSE); 2020: 209–220. https://doi.org/10.1145/3368089.3409715.

38. Tan L, Yuan D, Krishna G, Zhou Y. /* iComment: Bugs or Bad Comments? */. Symposium on Operating Systems Principles (SOSP); 2007: 145–158. https://doi.org/10.1145/1294261.1294276.

39. Blasi A, Goffi A, Kuznetsov K, et al. Translating Code Comments to Procedure Specifications. International Symposium on Software Testing and Analysis (ISSTA); 2018: 242–253. http://doi.acm.org/10.1145/3213846.3213872.

40. Breck E, Polyzotis N, Roy S, Whang SE, Zinkevich M. Data Validation for Machine Learning. Conference on Systems and Machine Learning (SysML); 2019. https://mlsys.org/Conferences/2019/doc/2019/167.pdf.

41. Habib A, Shinnar A, Hirzel M, Pradel M. Finding Data Compatibility Bugs with JSON Subschema Checking. International Symposium on Software Testing and Analysis (ISSTA); 2021: 620–632. https://doi.org/10.1145/3460319.3464796.