# Extracting Hyperparameter Constraints from Code

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

# EXTRACTING HYPERPARAMETER CONSTRAINTS FROM CODE

**Ingkarat Rak-amnouykit, Ana Milanova[1], Guillaume Baudart[2], Martin Hirzel, Julian Dolby[3]**
[1]Rensselaer Polytechnic Institute [2]Inria Paris and ENS-PSL University [3]IBM Research

## ABSTRACT

Machine-learning operators often have correctness constraints that cut across multiple hyperparameters and/or data. Violating these constraints causes runtime exceptions, but they are usually documented only informally or not at all. This paper presents a weakest precondition analysis for Python code. We demonstrate our analysis by extracting hyperparameter constraints for 45 sklearn operators. Our analysis is a step towards safer and more robust machine learning.

## 1 INTRODUCTION

To use machine-learning (ML) operators, data scientists must configure their *hyperparameters*, usually in the form of constructor arguments. For example, in sklearn (Buitinck et al., 2013), the `StandardScaler` operator has hyperparameters `with_mean` and `with_std`, and the `LogisticRegression` operator has hyperparameters `dual`, `solver`, `penalty`, etc. Incorrect hyperparameter configurations cause crashes, slowdowns, or sub-optimal accuracy. But configuring hyperparameters correctly is often not easy due to *hyperparameter constraints*. For example, `StandardScaler` does not allow `with_mean` to be true if the input data is sparse, and `LogisticRegression` does not allow `dual` to be true unless `solver=="liblinear"` and `penalty=="l2"`. We need a reliable formal specification of these constraints for dynamic precondition checks, static verifiers, or pruning the search spaces of automated hyperparameter tuning.

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 (Rak-amnouykit et al., 2020), they are also not expressive enough for constraints with logical implications across multiple hyperparameters, or across hyperparameters and data. Hyperparameter tuning tools, such as auto-sklearn (Feurer et al., 2015) or hyperopt-sklearn (Komer et al., 2014), come with search space specifications. But writing those specifications by hand is tedious and error-prone: for example, they take up 25 KLOC of Python in auto-pandas (Bavishi et al., 2019). Therefore, these search space specifications 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 verification.

One might be tempted to turn to natural-language documentation for hyperparameter constraints (Baudart et al., 2020b). But even though popular packages like sklearn have high-quality documentation, this is at most semi-formal and not always reliable. The code may raise an undocumented exception. For example, using the techniques in this paper, we found that sklearn's `ExtraTreesClassifier` raises an exception if `bootstrap==False` and `oob_score==True`. But the documentation did not mention this constraint; we submitted a pull request (github.com/scikit-learn/scikit-learn/pull/19444), which the developers confirmed and merged. As another example, the documentation for `GradientBoostingRegressor` describes the `alpha` hyperparameter with "Only if `loss=='huber'` or `loss=='quantile'`." One might think this is a correctness constraint, but in fact, the code does not raise an exception for it. A reliable formal specification can help debug the documentation.

This paper presents a static analysis for extracting hyperparameter constraints from code implementing machine-learning operators. We focus on Python and sklearn (Buitinck et al., 2013), since sklearn is the most widely-used ML framework today. Our analysis returns logic formulas with constraints over at least two hyperparameters or hyperparameters and data. We can use these as preconditions for dynamic checking at the interface, which is friendlier than raising an exception from deep within the implementation. For better error messages, the formulas are factored 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 verifiers of client code.

We ran our analysis on 45 ML operators from sklearn. On the 30 operators with non-trivial constraints, it achieved 82.9% precision and 99.8% recall. Overall, we hope that our interface specifications make machine-learning libraries more reliable and explainable, and serve as a step towards better formal verification of machine learning systems.

## 2 WEAKEST PRECONDITION ANALYSIS

The analysis is standard backwards reasoning, also known as weakest precondition reasoning (Hoare, 1969; Leino, 2005; Barnett et al., 2005), adapted for Python and extended with soundness reasoning about the generated precondition. The analysis starts from a `raise` exception statement in the code, then computes the precondition that must hold at the start of the enclosing function to prevent the exception being raised. At each step of backward reasoning, our analysis computes WP$(stmt, Q, S) \mapsto (Q', S')$: given a Python statement, a postcondition $Q$, and a soundness flag $S$, our analysis returns a pair $(Q', S')$ of a precondition $Q'$ and its soundness flag $S'$.

WP(`raise E`, $Q$, $S$) handles Raise statements:
**return** (`False`, `True`)

At a blank raise statement, an exception is certain, so the precondition for not raising it is $Q' =$ `False` and its soundness flag is $S' =$ `True`.

WP(`x=RHS`, $Q$, $S$) handles Assignment statements:
$Q' \leftarrow Q$[`RHS/x`]
**return** $(Q', S \wedge mod(\texttt{RHS}) \cap read(Q') = \emptyset)$

The precondition $Q'$ results from the substitution of left-hand side `x` with `RHS`. If the set of locations *modified* by RHS and the set of locations *read* by $Q'$ are disjoint, then $Q'$ is sound, meaning that $Q'$ evaluates to true iff after the execution of `x=RHS`, $Q$ evaluates to true, similarly to O'Hearn et al. (2001). Otherwise, $S'$ is false, meaning that precondition $Q'$ is potentially unsound.

WP(`if E: Seq1 else: Seq2`, $Q$, $S$) handles If statements:
$(Q_1, S_1) \leftarrow$ WP(`Seq1`, $Q, S$)
$(Q_2, S_2) \leftarrow$ WP(`Seq2`, $Q, S$)
$Q' \leftarrow (\texttt{E} \Rightarrow Q_1) \wedge (\texttt{not E} \Rightarrow Q_2)$
**return** $(Q', S_1 \wedge S_2 \wedge mod(\texttt{E}) \cap read(Q') = \emptyset)$

The precondition formula is standard. It is sound if (1) $Q$ is sound, (2) neither Seq1 nor Seq2 contain statements that invalidate the soundness, and (3) `E` has no effect on $Q_1$ or $Q_2$.

WP(`other`, $Q$, $S$) handles Other statements:
**return** $(Q,$ `False`)

Other statements are Python statements that do not match the syntax of the core subset specified above. These include `while`, `delete`, `try`, and the rest of the Statement nodes specified by the Python AST, as well as Assignments whose left-hand-side is not a variable or an attribute of self, e.g., `self.bootstrap`. The code for Other propagates $Q$ as is, however, it sets the soundness flag to `False`, since the statement may have affected $Q$. Our implementation does propagate exceptions raised from within `for` loops; when propagating backwards past a `for` loop, it is treated as Other.

We illustrate with the example from `StandardScaler` we mention in the introduction. Our analysis infers the sound precondition `issparse(X)` $\Rightarrow$ `not with_mean`:

```
{(( issparse (X) ⇒ not with_mean) and (not issparse(X) ⇒ True), True)} equiv. to {(issparse(X) ⇒ not with_mean, True)}
if issparse (X):
  {((with_mean ⇒ False) and (not with_mean ⇒ True), True)} equiv. to {(not with_mean, True)}
  if with_mean:
    {(False,True)}
    raise ValueError("Cannot center sparse matrices: pass 'with_mean=False' instead.")
  {(True,True)}
{(True,True)}
```

There are two sources of unsoundness: 1) the disjointness check fails and 2) Other Python statements such as `while` loops. Currently we assume that the disjointness check always returns true, which usually holds in practice. (We plan to implement known static analyses that verify disjointness.) A large percentage of preconditions are reported sound meaning that our handling of a core subset of Python is sufficient for this problem. As a concrete example, the analysis infers eight preconditions in `ExtraTreesClassifier`. *All* are reported sound. We make some additional assumptions. First, we infer the set of hyperparameters by examining constructor arguments, and assume that the code follows sklearn conventions and stores them as fields of the same name in the operator object. We treat syntactic occurrences of `X` and `y` as the only references to the input data. Our results indicate that, like the disjointness check, these assumptions hold true for sklearn. Second, we examine only exceptions that occur in the operator file, which our experiments indicate is insufficient. There were a number of runtime exceptions that our analysis did not catch as they were raised by imported code. In future work, we will extend the analysis with interprocedural and inter-module analysis.

## 3 Experimental Results

**Methodology.** For each operator, we start with carefully crafted schemas for individual hyperparameters that do not involve other hyperparameters or data. Sampling from the domain of these schemas, we generate 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__` function with the hyperparameter configuration, then calling its `fit` function 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 thrown and it passes otherwise.

The results from the dynamic exceptions are our ground truth. It is possible that some exceptions might not be covered. We translate weakest precondition constraints from the analysis into JSON schema (Pezoa et al., 2016) format and use schema validation to check the hyperparameter configuration against the schema. For example, a trial is a false positive if it fails but its hyperparameter configuration is valid against the schema from the analysis. As another example, a trial is a true negative if it fails and it is invalid against the schema as well.

**Research Question 1: How well does our weakest precondition analysis work?** We evaluate our analysis on 45 sklearn operators. Table 1 shows performances on 30 operators that have failed trials, excluding 15 operators where all trials succeeded (for those 15 operators, the analysis has 100% precision and 100% recall). For example, the constraints of `RandomForestClassifier` precisely reject 558 hyperparameter configurations from the failed trials. Currently, the analysis is intraprocedural and it misses exceptions from imported modules. In `LogisticRegression`, the analysis correctly identifies 594 true positives and 1,368 true negatives across 6 weakest precondition constraints. The remaining 38 trials are false positives where the runtime exceptions are raised by imported code. Overall, our analysis outperforms other methods, with an average F-score of 88.6.

**Research Question 2: How effective are hand-written constraints extracted from the documentation?** We also validate hyperparameter configurations against the hand-written constraints from the Lale open-source project (Baudart et al., 2020a). The hand-written constraints are extracted by careful examination of the documentation. Out of 30 operators with failed trials, 12 have logical hand-written constraints (for the 18 operators in the table with no such constraints, schema validation always passes.) Table 1 shows that performance of the hand-written constraints is slightly worse. This is mainly because they leave out constraints that appear in the code as exceptions but are missing from documentation. Similarly to the weakest precondition, hand-written constraints 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 negatives. Automatic weakest precondition analysis can be used to extract precise hyperparameter constraints and improve documentation.

**Research Question 3: How does our approach compare against related work?** Baudart et al. (2020b) 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 logical constraints.

| | Weakest Precondition | | | Hand-Written Constraints | | | NL Docstrings | | |
|---|---|---|---|---|---|---|---|---|---|
| | F-score | precision | recall | F-score | precision | recall | F-score | precision | recall |
| StandardScaler | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 85.7 | 75.0 | 100.0 |
| RobustScaler | 100.0 | 100.0 | 100.0 | 85.7 | 75.0 | 100.0 | 85.7 | 75.0 | 100.0 |
| ExtraTreesRegressor | 100.0 | 100.0 | 100.0 | 84.1 | 72.6 | 100.0 | 84.1 | 72.6 | 100.0 |
| ExtraTreesClassifier | 100.0 | 100.0 | 100.0 | 84.0 | 72.4 | 100.0 | 84.0 | 72.4 | 100.0 |
| RandomForestRegressor | 100.0 | 100.0 | 100.0 | 84.0 | 72.4 | 100.0 | 84.0 | 72.4 | 100.0 |
| RandomForestClassifier | 100.0 | 100.0 | 100.0 | 83.8 | 72.1 | 100.0 | 83.8 | 72.1 | 100.0 |
| SGDRegressor | 99.7 | 99.5 | 100.0 | 99.7 | 99.5 | 100.0 | 56.1 | 99.1 | 39.1 |
| GradientBoostingRegressor | 99.6 | 99.3 | 100.0 | 99.6 | 99.3 | 100.0 | 68.3 | 99.2 | 52.1 |
| QuantileTransformer | 99.1 | 98.3 | 100.0 | 99.1 | 98.3 | 100.0 | 99.1 | 98.3 | 100.0 |
| SGDClassifier | 98.3 | 96.7 | 100.0 | 98.3 | 96.7 | 100.0 | 30.1 | 94.5 | 17.9 |
| Ridge | 98.1 | 100.0 | 96.4 | 90.4 | 82.4 | 100.0 | 90.4 | 82.4 | 100.0 |
| RidgeClassifier | 98.1 | 100.0 | 96.4 | 90.4 | 82.4 | 100.0 | 90.4 | 82.4 | 100.0 |
| KNeighborsClassifier | 97.8 | 95.6 | 100.0 | 47.7 | 93.6 | 32.0 | 97.8 | 95.6 | 100.0 |
| KNeighborsRegressor | 97.8 | 95.6 | 100.0 | 47.7 | 93.6 | 32.0 | 97.8 | 95.6 | 100.0 |
| BaggingClassifier | 97.1 | 94.3 | 100.0 | 75.1 | 60.2 | 100.0 | - | - | - |
| LogisticRegression | 96.9 | 94.0 | 100.0 | 92.4 | 85.8 | 100.0 | 45.8 | 29.7 | 100.0 |
| RFE | 96.5 | 93.3 | 100.0 | 96.5 | 93.3 | 100.0 | - | - | - |
| PassiveAggressiveClassifier | 95.5 | 91.3 | 100.0 | 95.5 | 91.3 | 100.0 | 60.2 | 87.4 | 45.9 |
| Nystroem | 93.9 | 88.5 | 100.0 | 93.9 | 88.5 | 100.0 | 93.9 | 88.5 | 100.0 |
| FunctionTransformer | 93.8 | 88.3 | 100.0 | 75.5 | 83.9 | 68.6 | 93.8 | 88.3 | 100.0 |
| DecisionTreeRegressor | 90.0 | 81.8 | 100.0 | 90.0 | 81.8 | 100.0 | 90.0 | 81.8 | 100.0 |
| LinearSVC | 86.2 | 75.8 | 100.0 | 80.6 | 100.0 | 67.5 | 86.2 | 75.8 | 100.0 |
| NMF | 83.7 | 71.9 | 100.0 | 83.7 | 71.9 | 100.0 | 83.7 | 71.9 | 100.0 |
| PCA | 75.0 | 60.0 | 100.0 | 9.8 | 100.0 | 5.2 | 75.0 | 60.0 | 100.0 |
| GaussianNB | 66.7 | 50.0 | 100.0 | 66.7 | 50.0 | 100.0 | 66.7 | 50.0 | 100.0 |
| MinMaxScaler | 66.7 | 50.0 | 100.0 | 66.7 | 50.0 | 100.0 | 66.7 | 50.0 | 100.0 |
| QuadraticDiscriminantAnalysis | 66.7 | 50.0 | 100.0 | 66.7 | 50.0 | 100.0 | 66.7 | 50.0 | 100.0 |
| MissingIndicator | 65.8 | 49.0 | 100.0 | 65.8 | 49.0 | 100.0 | 65.8 | 49.0 | 100.0 |
| GradientBoostingClassifier | 64.4 | 47.5 | 100.0 | 64.4 | 47.5 | 100.0 | 64.4 | 47.5 | 100.0 |
| FeatureAgglomeration | 30.1 | 17.7 | 100.0 | 30.1 | 17.7 | 100.0 | - | - | - |
| average | 88.6 | 82.9 | 99.8 | 78.3 | 77.7 | 90.2 | 77.6 | 74.7 | 90.9 |

Table 1: Summary of results on operators that have failed trials.

Of the 30 operators, 27 operators have the natural-language documentation, but the technique only successfully extracts logical constraints for 7 operators. The remaining 23 operators do not raise exceptions and all their trials are either true positives or false positives. Our weakest precondition analysis outperforms the technique from (Baudart et al., 2020b) as shown Table 1.

## 4 Related Work

While backward reasoning and, more generally, verification condition generation have a long history, e.g., (Flanagan et al., 2002; Barnett et al., 2005) among other works, as far as we know, we are the first to apply these techniques on Python, whose rich dynamic semantics notoriously complicate static analysis. We posit that our novel "soundness" reasoning is a stride towards effective analysis of Python; while it is unrealistic to model every single construct, one can handle a core subset, assume that the rest of constructs have no effect, but still mark a constraint if affected by those other constructs. The programmer can manually verify soundness of marked constraints. In general, work on static analysis for Python is scarce. Ariadne (Dolby et al., 2018) explores static analysis of machine-learning libraries and outlines challenges to traditional static analysis techniques and Monat et al. (2020) present type analysis via abstract interpretation; we focus on the specific problem of extracting hyperparameter constraints. Finally, iComment for C (Tan et al., 2007) and jDoctor for Java (Blasi et al., 2018) have similar goal to ours — reconciling documentation with code and identifying issues with either of them. Our analysis outputs JSON schemas; Habib et al. (2021) describe a sub-schema checker for JSON.

## 5 Conclusions

This paper presents a static analysis for Python for extracting weakest precondition constraints. While the core analysis is standard, we extend it with a soundness flag and describe heuristics that make it effective on sklearn code. We automatically transform the analysis results to JSON schemas suitable for validation and search space pruning. In future work, we plan to make our analysis interprocedural.

## REFERENCES

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pp. 364–387, 2005. URL https://doi.org/10.1007/11804192_17.

Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. Lale: Consistent automated machine learning. In *KDD Workshop on Automation in Machine Learning (AutoML@KDD)*, August 2020a. URL https://arxiv.org/abs/2007.01977.

Guillaume Baudart, Peter Kirchner, Martin Hirzel, and Kiran Kate. Mining documentation to extract hyperparameter schemas. In *ICML Workshop on Automated Machine Learning (AutoML@ICML)*, July 2020b. URL https://arxiv.org/abs/2006.16984.

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: Neural-backed generators for program synthesis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019. URL https://doi.org/10.1145/3360594.

Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 242–253, 2018. URL http://doi.acm.org/10.1145/3213846.3213872.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: Experiences from the scikit-learn project, 2013. URL https://arxiv.org/abs/1309.0238.

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

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

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Jens Knoop and Laurie J. Hendren (eds.), *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pp. 234–245. ACM, 2002. URL https://doi.org/10.1145/512529.512558.

Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. Finding data compatibility bugs with JSON subschema checking. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2021.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. URL https://doi.org/10.1145/363235.363259.

Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *Python in Science Conference (SciPy)*, pp. 32–37, 2014. URL http://conference.scipy.org/proceedings/scipy2014/komer.html.

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

Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 17:1–17:29. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, pp. 1–19, 2001. URL https://doi.org/10.1007/3-540-44802-0_1.

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

Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. Python 3 types in the wild: A tale of two type systems. In *Dynamic Languages Symposium (DLS)*, pp. 57–70, 2020. URL https://doi.org/10.1145/3426422.3426981.

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