7th ICML Workshop on Automated Machine Learning (2020)

Mining Documentation to Extract Hyperparameter Schemas

Guillaume Baudart and **Peter D. Kirchner** and **Martin Hirzel** and **Kiran Kate**
IBM Research, New York, USA

Abstract

AI automation tools need machine-readable hyperparameter schemas to define their search spaces. At the same time, AI libraries often come with good human-readable documentation. While such documentation contains most of the necessary information, it is unfortunately not ready to consume by tools. This paper describes how to automatically mine Python docstrings in AI libraries to extract JSON Schemas for their hyperparameters. We evaluate our approach on 119 transformers and estimators from three different libraries and find that it is effective at extracting machine-readable schemas. Our vision is to reduce the burden to manually create and maintain such schemas for AI automation tools and broaden the reach of automation to larger libraries and richer schemas.

1. Introduction

Machine-learning practitioners use libraries of *operators*: reusable implementations of estimators (such as logistic regression, LR) and transformers (such as principal component analysis, PCA). Training an operator fits its *parameters* (learnable coefficients such as LR weights or PCA eigenvectors) to a dataset. Besides parameters, most operators also have *hyperparameters*: arguments that must be configured before training, such as the choice of LR solver or the number of PCA components. Python libraries for machine learning (ML) such as scikit-learn (Buitinck et al., 2013) tend to have good human readable documentation for hyperparameters. Unfortunately, this documentation is usually not easily machine readable. ML practitioners can configure hyperparameters either by hand or by using an HPO (automated hyperparameter optimization) tool such as hyperopt-sklearn (Komer et al., 2014) or auto-sklearn (Feurer et al., 2015), or the grid search or randomized search from scikit-learn. A *hyperparameter schema* specifies which hyperparameters are categorical and which continuous, which values or ranges are valid, and conditional hyperparameter constraints.

Python has recently emerged as the dominant ML language and many ML libraries adopt scikit-learn style conventions for interoperability (including PyTorch (sko), pandas (skl), Spark (spa), statsmodels (sta), and TensorFlow (ker)). This paper proposes and demonstrates an approach for mining hyperparameter schemas from the Python file implementing an ML operator. The approach, shown in Figure 1, mines the docstring and refines the resulting schema via dynamic analysis of the implementation. Using the Python file as a single source of truth simplifies maintenance when a new library version adds new features or dep-

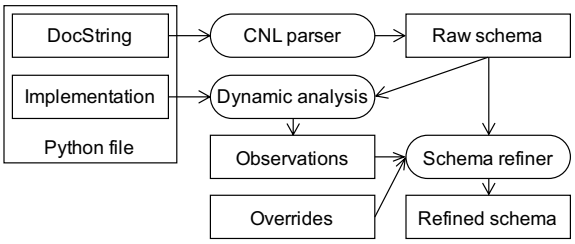

Figure 1: Overview of our mining approach.

recates old ones. Furthermore, since the Python file is written by library developers and the documentation is widely read by library users, it is a reliable source of truth.

Our approach outputs hyperparameter schemas in *JSON Schema*, which is a type description language for JSON documents (Pezoa et al., 2016). JSON Schema is widely adopted for web APIs, cloud management, and document databases, among other domains, and there are abundant public resources for learning and using it. JSON Schema is independent from specific AI automation tools and recent work has demonstrated that it can be converted to specifications for popular such tools (Baudart et al., 2020). We found JSON Schema to have just the right expressiveness for hyperparameters including categoricals and conditional constraints. Furthermore, we found JSON Schema easy to extend with additional meta-data such as distributions for continuous hyperparameters.

This paper makes the following contributions: (1) Mining Python docstrings to extract hyperparameter schemas including constraints. (2) Using dynamic analysis to obtain additional information about hyperparameters beyond the docstrings. (3) Reconciling hyperparameter metadata into a single machine-readable schema in JSON Schema format. We evaluate our approach on 119 automatically mined hyperparameter schemas for ML operators and 42 hand-curated schemas. We make both datasets publicly available (https://github.com/IBM/lale/tree/master/lale/lib/(autogen|sklearn)). Overall, we hope this paper contributes towards making HPO tools easier to use, more reliable, and more effective.

2. Problem Statement

This paper is about solving the problem of mining hyperparameter specifications from a Python docstring and turning them into a JSON Schema. To make things concrete, Figure 2 show an example input and the desired corresponding output of this mining problem.

The left side of Figure 2 shows an excerpt of class `sklearn.linear_model.LogisticRegression` with its docstring. A *docstring* is a string literal that documents a specific class or function definition. The HTML documentation for scikit-learn and other popular ML libraries is auto-generated from their docstrings. For this to work, the docstrings follow conventions understood by the HTML generation tool, in this case, the numpydoc extension (numpydoc maintainers, 2008) for Sphinx (Brandl, 2008). In other words, docstrings are written in a controlled natural language (CNL) (Kuhn, 2014): controlled, since they follow numpydoc conventions, and natural, since they are human-readable even before being converted into HTML. In practice, while docstrings suffice for HTML generation, they exhibit variability and typos that make schema extraction non-trivial.

This paper proposes an extractor that converts the docstring not to HTML but to JSON Schema. The right side of Figure 2 shows the schema for two categorical arguments `solver` and `penalty` and one continuous arguments `C`. Like most ML libraries, scikit-learn encodes categorical hyperparameters via Python string constants as opposed to Python enums, but only the values mentioned in the documentation are valid. The example also contains a conditional hyperparameter constraint: the value of `solver` implies which values are valid for `penalty`. We can express this implication by taking the logically equivalent form $\neg \mathit{premise} \vee \mathit{conclusion}$ and using the JSON Schema keywords `anyOf` and `not`.

```
class LogisticRegression:
  """Logistic Regression classifier.

  Parameters
  ------

  solver : str, {'linear', 'sag', 'lbfgs'}, \
           optional (default='linear').
    Algorithm for optimization.
    - Solvers 'sag' and 'lbfgs' support only l2.

  penalty : str, 'l1' or 'l2', default: 'l2'
    Norm used in the penalization.
    The 'sag' and 'lbfgs' solvers support
    only l2 penalties.

  C : float, default: 1.0
    Inverse regularization strength;
    must be a positive float.
    Like in support vector machines, smaller
    values specify stronger regularization
  """

  def __init__(self, solver='warn',
               penalty='l2', C=1.0, ...):
    self.solver = solver
    self.penalty = penalty
    self.C = C
    ...
```

```
{ '$schema':'http://json-schema.org/draft-04/schema#',
  'allOf': [
    { 'type': 'object',
      'additionalProperties': False,
      'required': ['solver', 'penalty', 'C'],
      'relevantToOptimizer': ['solver', 'penalty', 'C'],
      'properties': {
        'solver': {
          'description': 'Algorithm for optimization.',
          'enum': ['linear', 'sag', 'lbfgs'],
          'default': 'linear'},
        'penalty': {
          'description': 'Norm used in the penalization.',
          'enum': ['l1', 'l2'],
          'default': 'l2'},
        'C': {
          'description': 'Inverse regularization strength.',
          'type': 'number',
          'distribution': 'loguniform',
          'minimum': 0.0,
          'exclusiveMinimum': True,
          'default': 1.0,
          'minimumForOptimizer': 0.03125,
          'maximumForOptimizer': 32768},
      { 'description': 'Solvers sag and lbfgs support only l2.',
        'anyOf': [
          { 'type': 'object',
            'properties': {'solver':{'not':{'enum':['sag','lbfgs']}}}},
          { 'type': 'object',
            'properties': {'penalty':{'enum':['l2']}}}]}]}
```

Figure 2: Simplified excerpt of the scikit-learn code for the LogisticRegression estimator (left) and the corresponding hyperparameters schema (right).

3. Mining Docstrings

This section describes the CNL Parser component of the overview diagram in Figure 1. A CNL (controlled natural language) is a natural language (e.g., English) with some amount of structure (e.g., the input format for Sphinx and numpydoc, adopted by scikit-learn and other ML libraries). The CNL parser starts by reading the docstring from the Python file, such as the one on the left of Figure 2. It uses Sphinx and numpydoc to extract the docstring of methods `__init__` (class constructor), `fit` (for training), and `predict` or `transform` (for using an operator after training) of the operator class. Sphinx and numpydoc pre-parse this information into a list of argument tuples of the form `(name, short_desc, long_desc)` as well as descriptions of the return values of the methods. Given the list of argument tuples, the CNL parser uses two hand-crafted CNL grammars to extract per-argument schemas and inter-argument constraints, respectively.

Mining Argument Schemas. The CNL parser uses a grammar (see Figure 3 in the appendix) to extract per-argument schemas from the `short_desc` fields and inter-argument constraints that appear in the `long_desc`. The example of Figure 2 illustrates the main difficulties for mining and extraction. The parser needs to ignore noise such as whitespace, string quotes, or the trailing backslash (\). In addition, even within the same operator, the documentation uses different ways to express the same thing, e.g., enumerating values

in curly braces `{...}` vs. using `or`. Overall, these difficulties arise from the 'N' in CNL: docstrings use natural language. Noise is easy to handle using filtering during lexical analysis. For the other difficulties, our grammar takes advantage of the 'C' in CNL: docstrings use controlled language to the extent that they follow the conventions encouraged by Sphinx and numpydoc. The grammar thus specifies multiple syntactic alternatives to capture different ways to express the same thing (e.g., *enum* or *default*).

Mining Constraints. For inter-argument constraints, the CNL parser first extracts complete sentences from the long description. Next, it uses regular expressions to flag possible candidate constraints, for example, sentences containing the word 'only'. Then it parses each candidate using a grammar that captures common patterns (see Figure 4 in the appendix). Unfortunately, there is great variety in how docstrings express conditional hyperparameter constraints. Our grammar is only a first attempt to extract meaningful information. When the CNL parser fails to parse a sentence flagged as a potential constraint, it puts a placeholder into the schema with a `TODO` that a human can fill in later. Having mined both per-argument schemas and inter-argument constraints, the last step of the CNL parser is to assemble all the pieces into a single raw schema. The resulting JSON Schema is machine-readable and captures the information in a format suitable both for validation and for search.

4. Refining Mined Meta-Data

This section describes the Schema Refiner component of the overview diagram in Figure 1. This component uses dynamic analysis on the Python code to make additional observations, which it combines with heuristics and overrides to turn the raw schema from the CNL Parser into a refined schema for HPO tools.

Dynamic analysis for default values. Non-algorithmic defaults complicate the analysis of types and values. This occurs, for example, when an argument default is appropriated for purposes other than parameterization. To illustrate, it has become relatively commonplace within scikit-learn to advise users of upcoming changes in defaults for important arguments by setting the default value in the constructor's signature to `'warn'` (e.g., Figure 2 (left)) to trigger a warning message. Our dynamic analysis creates an instance by calling the constructor `__init__()` without passing any explicit arguments, then calls `fit` on the resulting instance — which might assign the argument to its actual algorithmic default value — and finally introspects the instance for these (possibly altered) default values and their types.

Dynamic analysis via runtime exception testing. We use the following techniques to harvest good values and filter bad values for constructor arguments.
Bad values: Defaulting all other arguments, we give a deliberately bad value to the argument under test and capture the exception. This exception text usually reports the bad value which is easily distinguished in the message. Frequently, the exception text also reports valid choices for the argument value, using a range of syntax that we can parse.
Greedy harvesting: We allow argument values that are valid for one operator to be tested for the same argument name in a different operator. This occasionally discovers valid values, particularly for under-documented classes.
Sampling: Defaulting all other arguments, an argument's range is sampled for testing for valid values. If categorical, all values are tested. Failed values are filtered out for the class.

The message received for deliberately false values can help to disambiguate the complaint in the case where it is not known a priori if the tested value is good or bad.
Bounds testing: With the caveat that some bounds may depend upon data and the values of other arguments, the bounds of continuous ranges can be tested individually for validity and exclusivity, which can be expressed in JSON Schema via e.g., `'exclusiveMinimum': True`.

Argument Overrides and Relevance to HPO. We provide for a dictionary of overrides that allow the automatically extracted types and ranges to be replaced with user-specified values, or for the parameter to be excluded from optimization. E.g., suppressing the `'mae'` choice for `'criterion'` on tree regressors because of prolonged fit times, or custom bounds for numeric parameters. The `ForOptimizer` suffix indicates that these are not hard constraints. This step also sets the `distribution` field (e.g. `loguniform`) and the `relevantToOptimizer` list, omitting irrelevant arguments such as verbosity.

5. Results

This sections measures the effectiveness of our extractor tool with three experiments. (1) We mined the schema of 119 operators from three different libraries: 115 from scikit-learn (Buitinck et al., 2013), 2 from XGBoost (Chen and Guestrin, 2016), and 2 from LightGBM (Ke et al., 2017). (2) We compared the schemas of 42 operators with manually curated schemas: 38 from scikit-learn, 2 from XGBoost, and 2 from LightGBM. (3) We used the generated schemas to find three-steps pipelines for 15 OpenML datasets with Lale (Baudart et al., 2020), an Auto-ML library that uses hyperparameter search spaces in JSON Schema.

Complete dataset.
Table 1 presents the results of the extractor executed on the complete dataset (see also Appendix B.1). For each category, we report the number mined by the CNL parser and the corrections made by the schema refiner (*parser*+*refiner*). For the constraints we report the number of valid constraints and the number of detected constraints (*valid*/*detected*). Overall, we were able to mine 94% of the 1,758 argument types (including the input/output schemas of the `fit`, `transform`, and `predict` methods of all the operators). We extracted a default value for 64% of the arguments but default values are not always relevant, e.g., for the input/output type of the `fit` or `predict` method. We found a valid range for 50% of the 790 relevant arguments, i.e., numeric arguments (`integer` or `number`) or `string` arguments used to captures `enum` values. Finally, we detected 120 constraints but only 43 were converted into valid JSON Schema.

Table 1: Summary of the auto-generated schemas.

	total	coverage	scikit-learn	xgboost	lightgbm
classes	119	1.00	115	2	2
arguments	1,867	1.00	1,686	88	93
types	1,758	0.94	1,606 (1,490 + 116)	77 (73 + 4)	75 (69 + 6)
default	1,204	0.64	1,090 (660 + 430)	49 (13 + 36)	65 (61 + 4)
range	399	0.50	339 (0 + 339)	37 (0 + 37)	23 (0 + 23)
constraints	43	0.36	43 /118	0 /0	0 /2

Curated dataset. Table 2 presents the results of the comparison (see also Appendix B.2). For this experiment, we focused on the arguments to the operator's constructor. The extractor correctly mined the type for 81% of the arguments and the default value for 97%.

The extractor also found a valid range for 81% of the 103 defined ranges in the curated set, and 75% of the distributions. Finally, the extractor detected 50 of the 65 constraints of the curated set. Among the detected constraints, 20 are converted into valid JSON Schema and 90% of these match the curated schemas.

Table 2: Auto-generated vs. curated schemas.

	reference	generated	match	precision	recall	F_1
arguments	452	452	452	1.00	1.00	1.00
types	452	399	367	0.92	0.81	0.86
defaults	452	441	438	0.99	0.97	0.98
ranges	103	83	83	1.00	0.81	0.89
distributions	166	125	125	1.00	0.75	0.86
constraints	65	20 (/50)	18	0.90	0.28	0.42

Auto-ML pipelines. To demonstrate the use of our schemas, we use Lale pipelines of the form `preprocessor >> feature_extractor >> classifier`. Then, we let AutoML automatically select each step from a predefined set of operators (see details in Appendix B.3) and tune its hyperparameters based on our extracted schemas. For comparison, we used auto-sklearn (Feurer et al., 2015) with the same resource constraints as a baseline: 1h of optimization time and a timeout of 6mn per trial. Note that in this comparison, both the framework and the hyperparameter schemas differ. The results show that Lale with our auto-generated schemas achieves similar accuracies as auto-sklearn, a state-of-the-art tool.

6. Related Work

The most closely related work is jDoctor, which mines javadoc comments to extract method pre- and post-conditions (Blasi et al., 2018). The results of mining are similar to schemas in that they can capture argument ranges and even some constraints. But jDoctor focuses on Java, whereas we focus on Python code without static type annotations and with string constants. Furthermore, jDoctor focuses on testing, whereas we focus on AutoML.

Our schema refiner uses dynamic analysis on Python code to augment the information extracted from docstrings. Fuzz testing, also known as *fuzzing*, is a well-established approach for finding software defects by generating random inputs (Miller et al., 1990). While our schema refiner is inspired by fuzzing, its goal is not to find defects but to extract schemas.

The primary contribution of this paper is the documentation mining, not the chosen output format. We could have used different formats to express hyperparameter schemas. Python 3 introduces optional type annotations that can be checked statically (van Rossum et al., 2014). Unfortunately, since Python 3 types lack intersection types, string constant types, and conditional constraints, they are less suitable for HPO. PCS is a file format for specifying hyperparameter schemas for the SMAC tool (Hutter and Ramage, 2015). PCS is well-suited for HPO and JSON Schema can be converted to PCS (Baudart et al., 2020).

7. Conclusion

This paper presents a tool that mines Python docstrings of ML libraries to extract hyperparameter schemas for HPO. The extracted schemas include names, types, defaults, and descriptions, ranges and distributions for continuous hyperparameters, enumerations of constants for categorical hyperparameters, and constraints for conditional hyperparameters.

References

tf.keras.wrappers.scikit_learn. URL https://www.tensorflow.org/api_docs/python/tf/keras/wrappers/scikit_learn. (Retrieved June, 2020).

sklearn-pandas. URL https://github.com/scikit-learn-contrib/sklearn-pandas. (Retrieved June, 2020).

skorch. URL https://github.com/skorch-dev/skorch. (Retrieved June, 2020).

spark-skearn. URL https://github.com/databricks/spark-sklearn. (Retrieved June, 2020).

statsmodels. URL https://github.com/statsmodels/statsmodels. (Retrieved June, 2020).

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)*, 2020. URL https://github.com/ibm/lale.

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)*, pages 242–253, 2018. URL http://doi.acm.org/10.1145/3213846.3213872.

Georg Brandl. Sphinx Python documentation generator, 2008. URL http://sphinx-doc.org/. (Retrieved June, 2020).

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.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 785–794, 2016. URL http://doi.acm.org/10.1145/2939672.2939785.

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)*, pages 2962–2970, 2015. URL http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.

Frank Hutter and Steve Ramage. Manual for SMAC version v2.10.03-master, 2015. URL https://www.cs.ubc.ca/labs/beta/Projects/SMAC/v2.10.03/manual.pdf. (Retrieved June, 2020).

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Conference on Neural Information Processing Systems (NIPS)*, pages 3146–3154, 2017. URL http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.

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

Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, 2014. URL http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00168.

Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of Unix utilities. *Communications of the ACM (CACM)*, 33(12):32–44, December 1990. URL http://doi.acm.org/10.1145/96267.96279.

numpydoc maintainers. numpydoc – Numpy's Sphinx extensions, 2008. URL https://numpydoc.readthedocs.io. (Retrieved June, 2020).

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

Guido van Rossum, Jukka Lehtosalo, and Lukasz Langa. PEP 484 – type hints, 2014. URL https://www.python.org/dev/peps/pep-0484/. (Retrieved June, 2020).

Appendix A. Mining Docstrings

A.1 Mining Argument Schemas

Sphinx and numpydoc pre-parse the documentation into a list of argument tuples of the form `(name, short_desc, long_desc)`. The CNL parser uses the grammar in Figure 3 to extract per-argument schemas from the `short_desc` fields.

The start symbol of the grammar, *start*, splits the docstring into three parts: the type, or a sequence of possible types (encoded in JSON using the `anyOf` keyword); an optional flag; and the default value.

A.2 Mining Constraints

For inter-argument constraints, the CNL parser first extracts complete sentences from the long description. Next, it uses a set of regular expression rules to flag possible candidate constraints, for example, sentences containing the word 'only'. On each candidate it discovers, the CNL parser uses the grammar in Figure 4.

Unfortunately, there is great variety in how docstrings express conditional hyperparameter constraints. Since our CNL grammar does not anticipate the syntax of this example, it cannot fully extract this constraint. However, it does detect the presence of some constraint, and it puts a placeholder into the raw JSON Schema with a `TODO` that a human domain expert can then fill in later to further enhance it.

Appendix B. Results

This section measures the effectiveness of our extractor tool on two datasets: complete and curated. The *complete dataset* comprises 119 operators from three different libraries: 115 from scikit-learn (Buitinck et al., 2013), 2 from XGBoost (Chen and Guestrin, 2016), and 2 from LightGBM (Ke et al., 2017). For XGBoost and LightGBM we considered both the regressor and the classifier. For scikit-learn we filter the classes revealed by `sklearn.utils.testing.all_estimators()` to obtain estimators and transformers. We exclude classes that are abstract, or that are meta-estimators. Further we examine their method resolution order, confirm the existence of `fit`, and `predict` or `transform`, and confirm their signatures. Finally, we exclude some classes known to be deprecated, e.g., `Imputer`, and some known to be intended only to be used by other classes, e.g., `ExtraTree`.

The *curated dataset* comprises 42 operators with manually curated schemas: 38 from scikit-learn, 2 from XGBoost, and 2 from LightGBM. Whereas the complete dataset allows us to evaluate the robustness and coverage of the tool overall, the curated dataset allows us to compare the extracted schemas against a ground truth for reference.

Finally, we used the generated schemas to find three-steps pipelines for 15 OpenML datasets with Lale and compare the results with with auto-sklearn.

B.1 Complete Dataset Evaluation

Table 1 presents the results of the extractor executed on the complete dataset. For each library, it reports the number of extracted arguments, types, default values, ranges, and constraints. In addition, Table 1 explicitly shows the contributions of the CNL parser

```
start     ::= seq optional default (. | ,)?
seq       ::= (type ,?)+ (or type)?

type      ::= int | integer | float | double | boolean | bool | string | str
            | None | Ignored | callable | dict | type
            | obj | array | enum

optional  ::= (, optional )?

default   ::= ,? (default (= | :)? val
            | ( default (= | :)? val )
            | val by default
            | or val ( default ))?

obj       ::= object | RandomState instance | returns an instance of self

array     ::= atype (shape)?
atype     ::= list | array | tuple | array_like | array-like
            | numpy array | sparse matrix | scipy.sparse | scipy sparse
            | {? atype (or | ,) atype }?
shape     ::= ,? of ? (shape | size )? =? vtuple (or shape)?

enum      ::= { val (,? or ? (an | a )? val)* }
            | (string | str) ,? enum
            | [? val (| val)+ ]?

vtuple    ::= (( | [) val (, val)* ,? (] | )) | None

val       ::= NAME | NUMBER
```

Figure 3: CNL grammar for parsing per-argument schemas.

```
start     ::= only when cond

only      ::= only (used | effective | compatible | significant | available | applies )?
when      ::= when | if | with | in | for

cond      ::= atom | cond (and | or) cond
atom      ::= NAME compare seq | the? seq NAME (is used)?
seq       ::= val ((, val)* ,? (and | or) val)?

compare   ::= == | = | > | < | >= | >= | is set to | is
val       ::= NUMBER | NAME
```

Figure 4: CNL grammar for parsing inter-argument constraints.

and the schema refiner for each category to illustrate the advantages of combining the two strategies.

Overall, we were able to mine 94% of the 1,867 argument types (including the input/output schemas of the `fit`, `transform`, and `predict` methods of all the operators). Missing argument types mostly come from unsupported or under-specified data structures, e.g., `object`, `dict`, or `callable`. Even when mined, it is not clear how an AI automation tool would instantiate such arguments during search.

We extracted a default value for 64% of the arguments but default values are not always relevant, e.g., for the input/output type of the `fit` or `predict` method. Table 1 shows that, even if default values are often documented, the schema refiner can extract a lot of additional information, e.g., via dynamic analysis.

Since ranges are not consistently documented, the range analysis is solely based on the schema refiner. We found a valid range for 50% of the 792 relevant arguments, i.e., numeric arguments (`integer` or `number`) or `string` arguments used to captures `enum` values. However, ranges are not required for all of these arguments. In fact we observe that, even in the curated schemas, ranges are only defined for a few arguments to produce valid search spaces for hyperparameter search tools.

Finally, we detected 120 constraints but only 43 were converted into valid JSON Schema. The remaining 77 generate `TODO` warnings that can be manually inspected.

B.2 Curated Dataset Evaluation

Next, we compare the result of the extractor with a set of manually curated schemas. Results are presented in Table 2. Compared to Table 1 we focus on the arguments to the operator's constructor (`__init__`), leaving aside the arguments and return values of its `fit`, `predict`, or `transform` methods For each category we report the reference number, the generated number, and the number of matches between the two sets. We also report the corresponding precision, recall, and F_1 score.

The extractor is able to detect all the 452 arguments, which indicates that all the arguments are consistently documented across the three libraries. In the curated set, all arguments have a correct type and a default value. The extractor correctly mined the type for 81% of the arguments and the default value for 97%. The extractor also found a valid range for 81% of the 103 defined ranges in the curated set. We categorized a range as valid if the extractor returns an interval included in the range defined in the curated schema. Compared to the complete dataset, these counts are relatively low because the curated dataset includes the XGBoost and LightGBM operators that are both more complex and less documented.

Mismatches between extracted and curated default values are due to values that cannot be represented in JSON Schemas: the default value of the `missing` argument of the XGBoost operators and the `missing_values` of `SimpleImputer` is `nan`, which is replaced by `None` in the curated schemas. Additionally, our extractor found inconsistencies between the documentation and the code. For instance, the documentation for the `categories` argument of `OneHotEncoder` is:

```
categories : 'auto' or a list of lists/arrays of values, \
  default='auto'.
```

but the `OneHotEncoder.__init__` method gives a default value `categories=None`.

Arguments types are often complex *union* types allowing multiple choices for one argument. For example, `n_jobs` can often be either `None` or an `integer`. To further investigate the discrepancies between generated and curated types, we analyzed the numbers of *type values* and *enum values*. The *type values* analysis reports the number of *terminal* types found in the schemas, i.e, `boolean`, `integer`, `number`, `string`, or `enum`. The *enum values* analysis reports the number of members found in each `enum` list.

	reference	generated	match	precision	recall	F_1
type values	631	611	525	0.86	0.83	0.85
enum values	426	269	238	0.88	0.56	0.68

We observe that for both analyses the precision is relatively high: 86% for type values and 88% for enum values, which suggests that extracted data are mostly correct. However, for enum values the generated number is significantly lower than the curated number: the extractor only found 56% of the enum values in the curated dataset.

This is mostly due to arguments that are under-specified as `string` in the documentation and for which the schema refiner can not find a suitable enumeration. For example, the documentation of the `criterion` argument of `GradientBoostingClassifier` is:

```
criterion: string, optional (default="friedman_mse")
```

but the valid enumeration is `['friedman_mse', 'mse', 'mae']`. These under-specifications are relatively common when the enum value is only one possible choice in a complex union type. For example, the documentation of the `max_features` argument of `DecisionTreeClassifier` is:

```
max_features: int, float, string or None, optional \
  (default=None)
```

but again the valid enumeration is `['auto', 'sqrt', 'log2']`.

Finally, the extractor detected 50 of the 65 constraints of the curated set. Among the detected constraints, 20 are converted into valid JSON Schema and 90% of these match the curated schemas. The mismatches are due to complex constraints that are merged in the curated schemas. For instance, the description of the `power_t` argument of `MLPClassifier` contains:

```
It is used in updating effective learning rate when
the learning_rate is set to 'invscaling'.
Only used when solver='sgd'.
```

which is captured in the curated schemas as: "`power_t` can only differ from its default value 0.5 if `solver == 'sgd'` and `learning_rate == 'invscaling'`", or in JSON Schema:

```
'anyOf': [
    {'type': 'object',
     'properties': {'power_t': {'enum': [0.5]}}},
    {'type': 'object',
     'properties': {
        'learning_rate': {'enum': ['invscaling']},
        'solver': {'enum': ['sgd']}}}]}
```

but the extractor is only able to extract the second condition on the solver (`solver == 'sgd'`).

The results for the constraints show that, even if we are able to flag most of the constraints, our CNL parser is not the best tool to extract meaningful information from the constraint candidate. The language used to express the constraints is far less constrained than the one used for the type description. An obvious direction for future work is thus to try classic natural language understanding techniques.

B.3 AutoML Pipelines

The selected datasets comprise 5 simple classification tasks (test accuracy $> 90\%$ in all our experiments) and 10 relatively complex tasks (test accuracy $< 90\%$). For all the tasks we start from the same three-step pipeline with both generated — lale-gen in Table 3 — and curated schema — lale-cur in Table 3. For comparison, we used auto-sklearn (Feurer et al., 2015) — autoskl in Table 3 — as a baseline. All tasks were configured with the same resource constraints: one hour of optimization time and a timeout of six minutes per trial. Lale then uses the search spaces defined in the schemas, the topology of the pipeline, and off-the-self optimizers such as Hyperopt (Komer et al., 2014), to find the best candidate.

```
preprocessors       = [ NoOp, MinMaxScaler, StandardScaler, Normalizer, RobustScaler]
features_extractors = [ NoOp, PCA, PolynomialFeatures, Nystroem]
classifiers         = [ GaussianNB, GradientBoostingClassifier, KNeighborsClassifier,
                        RandomForestClassifier, ExtraTreesClassifier,
                        QuadraticDiscriminantAnalysis,  PassiveAggressiveClassifier,
                        DecisionTreeClassifier, LogisticRegression,  XGBClassifier,
                        LGBMClassifier, SVC ]

lale_pipe = make_pipeline( make_choice(*preprocessors),
                           make_choice(*features_extractors),
                           make_choice(*classifiers) )
```

For each experiment, we used a $66\% - 33\%$ validation-test split, and a 5-fold cross validation on the validation split during optimization. Experiments were run on a 32 cores (2.0GHz) virtual machine with 128GB memory. Table 3 shows the accuracy results (mean and standard deviation across 5 independent runs) and the number of runs for each experiments (where "ok" indicates a successful run and "ko" indicates an aborted run).

We observe that Lale with our auto-generated schemas achieves accuracies (88.1%) that are comparable to auto-sklearn (88.0% with auto) and Lale with curated schemas (88.5%). However, the number of aborted runs show that side-constraints play a key role during the optimization process. Only 4.0% of the runs using curated schemas were aborted, compared to 21.7% with generated schemas (and 6.7% for auto-sklearn).

Table 3: Accuracy results for the OpenML classification tasks

	autoskl				lale-gen				lale-cur			
	Precision		Runs		Precision		Runs		Precision		Runs	
dataset	mean	std	ok	ko	mean	std	ok	ko	mean	std	ok	ko
australian	85.09	(0.39)	527.8	(24.8)	85.35	(0.45)	416.8	(12.4)	86.14	(1.02)	146.0	(29.2)
blood	77.89	(1.24)	688.4	(42.2)	75.63	(1.61)	351.6	(9.6)	76.28	(4.57)	213.8	(43.8)
breast-cancer	73.05	(0.52)	758.6	(45.2)	72.63	(2.40)	552.0	(16.2)	72.00	(1.43)	202.4	(40.4)
car	99.37	(0.09)	461.6	(9.0)	99.16	(0.20)	138.4	(9.6)	99.19	(0.49)	123.8	(28.4)
credit-g	76.61	(1.07)	328.0	(22.4)	76.30	(0.87)	223.4	(11.6)	74.73	(0.68)	109.2	(24.4)
diabetes	77.01	(1.18)	545.0	(27.4)	75.83	(0.81)	384.2	(10.2)	76.77	(1.61)	217.8	(43.8)
hill-valley	99.45	(0.87)	343.0	(49.6)	99.65	(0.30)	113.0	(4.0)	99.60	(0.20)	79.4	(18.2)
jungle-chess	88.06	(0.22)	28.2	(8.2)	86.66	(1.06)	47.4	(7.0)	90.29	(0.00)	85.8	(19.4)
kc1	83.79	(0.28)	323.4	(21.4)	83.19	(0.27)	175.0	(6.2)	83.28	(1.10)	100.4	(22.0)
kr-vs-kp	99.70	(0.04)	176.2	(16.0)	99.34	(0.15)	79.0	(8.0)	99.55	(0.22)	61.0	(17.8)
mfeat-factors	98.70	(0.07)	114.8	(19.8)	97.33	(0.51)	62.0	(12.8)	97.85	(0.22)	46.0	(17.2)
phoneme	90.31	(0.35)	292.8	(3.6)	88.62	(0.46)	198.4	(4.6)	88.93	(0.49)	129.2	(24.0)
shuttle	87.27	(10.3)	70.8	(8.6)	99.92	(0.01)	49.0	(8.4)	99.94	(0.01)	60.6	(16.0)
spectf	87.93	(0.77)	493.4	(44.4)	88.10	(2.58)	276.6	(6.2)	88.10	(2.14)	134.0	(26.2)
sylvine	95.42	(0.19)	159.8	(12.2)	94.46	(0.19)	128.2	(0.4)	95.14	(0.14)	80.0	(18.2)