Pipeline Combinators for Gradual AutoML

Anonymous Author(s)
Affiliation
Address
email

Abstract

Automated machine learning (AutoML) can make data scientists more productive. But total automation misses out on human intuition. Instead, data scientists often prefer gradual automation, where a given AutoML search explores only certain choices, while the user controls the rest. With state-of-the-art tools, gradual AutoML is cumbersome, requiring large non-compositional code changes. Combinators are a powerful concept from functional programming that yields concise compositional code. This paper introduces a small set of orthogonal combinators for composing machine-learning operators into pipelines. It describes a translation scheme from pipelines and associated hyperparameter schemas to search spaces for AutoML optimizers. On that foundation, this paper presents Lale, an open-source sklearn-compatible AutoML library, and evaluates it with a user study.

1 Introduction

Automated machine learning can improve data scientist productivity by lowering the barrier to entry for novices as well as reducing the tedium for experts. With *total* automation, the user just provides data and the tool does the rest. While this yields a convenient starting point, it gives users little control. Instead, *gradual* automation, where the user tweaks some choices by hand, leverages user expertise for better outcomes and wastes less compute time. Unfortunately, gradual automation is not easy with existing AutoML tools and their programming interface. Code for automated machine learning tends to look different from that for manual machine learning, preventing a gradual transition between the two. Furthermore, code changes for gradual automation tend to be large and non-compositional. The programming model is often coupled to a given optimizer. And finally, tools often support controlling some aspects of the search only with a much lower-level programming interface, if at all.

Combinators were first discovered by Schönfinkel in the 1920s in an attempt to find elementary building blocks of logic [28]. As functions that compose functions, they play an important role in functional programming. For instance, Lisp provides the combinators `map` and `reduce` [30], which inspired the MapReduce programming model [9, 16]. Combinators enable a *tacit* programming style, where data remains unnamed, keeping function compositions concise and minimizing edits when re-arranging them. This paper argues that sklearn's `make_pipeline` and `make_union` [6] are combinators: they compose machine-learning operators into pipelines without naming datasets such as `X` or `y` during composition. Combinators enable the two primary execution modes, prediction and training, of sklearn pipelines. This paper streamlines these combinators with infix notation, adds a choice combinator, and adds AutoML search as a third execution mode.

Enabling search as a third execution mode requires a translation from combinators to AutoML search spaces. To this end, this paper describes a novel translation scheme, which takes a combinator-based pipeline and generates a search space suitable for a given AutoML optimizer. The translation first normalizes and combines the hyperparameter *schemas* of the operators in the pipeline, which specify the space of hyperparameter values. Next, depending on the optimizer at hand, it flattens and discretizes the search space. This translation is essential to bridge from the compositional and tacit

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

combinators syntax to an optimizer's syntax, and may re-arrange components and introduce (possibly mangled) names as needed. The translation scheme also supports round-tripping back to combinators, needed for evaluating and understanding results as well as for iterative refinement.

This paper introduces Lale, a Python AutoML library implementing the combinators along with the translation scheme. Lale enjoys active use both in the open-source community and in a major industrial product. It is sklearn-compatible, giving users access to many operators and familiar functionality. Lale is gradual, letting users specify only what they want while reusing and automating the rest. For instance, Lale comes with an extensive library of reusable hyperparameter schemas for many popular operators, so users rarely need to write their own schemas; but it also makes it easy to customize schemas when desired. There are Lale optimizer backends for Hyperopt [5] (with 4 solvers tpe, atpe, rand, and anneal); sklearn's GridSearchCV and HalvingGridSearchCV [6]; ADMM [19] (with 5 continuous and 5 combinatorial solvers); SMAC [13]; and Hyberband [18].

The contributions of this paper are:

- Combinators for gradual automated machine learning.
- A translation scheme for mapping combinators to a broad variety of optimizers.
- The Lale library and a user study for validating combinators and gradual automation.

We hope this work presents a step towards democratizing AutoML while keeping the user in control.

2 Pipeline Combinators

This paper proposes three combinators to compose machine-learning operators into pipelines. To support hyperparameter tuning, each operator has an associated hyperparameter schema. Thus, when combinators compose operators, then under the hood, that implies weaving together the associated hyperparameter schemas. This section describes the syntax both for combinators and for schemas, leaving it to Section 4 to describe the translation that weaves them together for AutoML.

Combinators. The three combinators are `>>`, `&`, and `|`. The *pipe combinator*, `p >> q`, connects the output of `p` to the input of `q`, similar to sklearn's `make_pipeline`. The *and combinator*, `p & q`, composes `p` and `q` side-by-side without creating dataflow between them. It is similar to sklearn's `make_union`, except that `make_union(p, q)` implicitly concatenates the output features of `p` and `q` whereas `p & q` does not. Instead, if users want to combine features after `&`, they have to explicitly add an operator for that. Making this explicit makes code more understandable and also provides flexibility to pick different data-combining operators when desired, for instance, vertical stacking vs. relational joining. The *or combinator*, `p | q`, introduces a choice between `p` and `q`. While it has no direct equivalent in sklearn, it can be encoded in sklearn's `GridSearchCV`, as well as in other optimizers such as Hyperopt.

pipeline	::=	*individualOp* | *pipe* | *and* | *or*
individualOp	::=	*NAME* | *NAME* `(` *hyperParam** `)`
hyperParam	::=	*NAME* `=` *expression*
expression	::=	*VALUE* | *pipeline*
pipe	::=	*pipeline* `>>` *pipeline*
and	::=	*pipeline* `&` *pipeline*
or	::=	*pipeline* `|` *pipeline*

Figure 1: Pipeline syntax. Non-terminals are in lower-case *italics*, literal tokens (`(`, `)`, `=`, `>>`, `&`, `|`) are in `type-writer` font, other terminals are in all-caps *ITALICS*, and the meta-syntax (::=, |, *) indicates rules, choice, and Kleene star.

```
pre_n = (Project(columns={"type":"number"})
        >> (PCA | NoOp))
pre_s = (Project(columns={"type":"string"})
        >> (OrdinalEncoder|OneHotEncoder))
pplan = ((pre_n & pre_s)
        >> ConcatFeatures
        >> AdaBoostClassifier(
              base_estimator=(J48 | LR)))
```

Figure 2: Pipeline example with two preprocessing sub-pipelines. Features from both are concatenated and piped to a boosted ensemble, whose base estimator is a choice of classifiers.

Figure 1 shows the syntax for combinator-based pipelines. The start symbol, *pipeline*, can be an individual operator or it can nest other pipelines by using one of the three combinators. An *individualOp* has a name and an optional configuration for zero or more hyperparameters. For example, in Figure 2, the operators `Project` and `AdaBoostClassifier` are explicitly configured with hyperparameters, whereas the operators `PCA`, `NoOp`, etc. are not. When the user does not manually configure hyperparameters, then AutoML automatically tunes them. A *hyperParam* configuration uses an *expression*, which can be a simple Python value or a nested pipeline. For example, in Figure 2, `Project` is configured with

a simple Python dictionary, whereas `AdaBoostClassifier` is configured with a nested pipeline. An operator that takes another operator as an argument is *higher-order*, and besides `AdaBoostClassifier`, there are plenty of other examples, such as sklearn's `RFE`, `OneVsRestClassifier`, or `ColumnTransformer`.

What makes the combinator-based pipeline syntax powerful is that everything nests. A pipeline is itself an operator in the sense that it implements the methods `fit` and `transform` or `predict`. The syntax is concise, as it focuses just on operators and is tacit about data. The Lale library implements this syntax as pure Python code. That has the advantage that users need not learn a new programming language, and can instead make full use of Python's existing features. For example, Figure 2 uses a Python assignment (`pre_n = ..`) to improve readability. Furthermore, existing Python tooling just works, such as auto-completing editors, syntax and type checkers, and interactive notebooks.

Execution modes. Pipelines have three execution modes: predict or transform; fit; and AutoML search. The first two modes come from sklearn. Their semantics are based on the dataflow graph of a pipeline, which is a directed acyclic graph whose vertices are operators and whose edges are given by the `>>` combinator (note that `&` and `|` do not induce dataflow edges, and pipelines with `|` only support the third execution mode). The first execution mode of a pipeline, predict or transform, simply invokes predict or transform on the vertices in topological order, guaranteeing that the results from all predecessors of an operator are available before that operator fires. The second execution mode, fit, also processes vertices in topological order, calling first fit and then transform or predict. The result of fit on a pipeline is a new pipeline that is the same as the original except that all operators are replaced by trained versions, i.e., their learned coefficients are bound. Higher-order operators have the flexibility to invoke the methods of nested operators as needed for the outer operator.

The third execution mode of a pipeline, AutoML search, is implemented by translating the planned pipeline to a search space for an AutoML optimizer. The search space encodes joint operator selection and hyperparameter tuning. Given a planned pipeline such as `pplan` from Figure 2, users access this execution mode by invoking the `auto_configure` method:

```
best_found = pplan.auto_configure(
    train_X, train_y, optimizer=Hyperopt, cv=3, max_opt_time=300, max_eval_time=30)
```

The result (in this case `best_found`) is the trained pipeline that yielded the best score (here, cross-validated with `cv=3`) during the search. Section 4 discusses the details of the translation.

Schemas. Besides combinators, which specify a space of pipelines to search, AutoML also needs hyperparameter schemas, which specify a space of hyperparameter values to search. Here, we adopt the *value-set* definition of a schema (or type) as a space (or set) of values [24]. While data scientists frequently write code with combinators to exercise control, they write schemas less often. As of today, Lale includes schemas for 216 operators, sufficient for many common machine learning tasks.

schema	::=	*enum* | *range* | *object* | *disj* | *conj* | *neg*
enum	::=	`[` *VALUE*$^+$ `]`
range	::=	`(` *NUM* `..` *NUM* `)` *distr*$^?$
distr	::=	"uniform" | "loguniform"
object	::=	`dict {` (*NAME* : *schema*)* `}`
disj	::=	*schema* $\vee$ *schema*
conj	::=	*schema* $\wedge$ *schema*
neg	::=	$\neg$ *schema*

Figure 3: Schema syntax.

$$
\begin{aligned}
\mathit{PCA} &: \mathrm{dict}\{N\colon (0..1) \vee [\mathit{mle}]\} \\
\mathit{J48} &: \mathrm{dict}\{R\colon [\mathit{true}, \mathit{false}], C\colon (0..0.5)\} \wedge \\
&\quad (\mathrm{dict}\{R\colon [\mathit{true}]\} \Rightarrow \mathrm{dict}\{C\colon [0.25]\}) \\
\mathit{LR} &: \mathrm{dict}\{S\colon [\mathit{linear}, \mathit{sag}, \mathit{lbfgs}], P\colon [\mathit{l1}, \mathit{l2}]\} \wedge \\
&\quad (\mathrm{dict}\{S\colon [\mathit{sag}, \mathit{lbfgs}]\} \Rightarrow \mathrm{dict}\{P\colon [\mathit{l2}]\})
\end{aligned}
$$

Figure 4: Schema examples for 3 operators (simplified): sklearn's `PCA` and `LogisticRegression` and Weka's `J48`. We write $s \Rightarrow t$ for $(\neg s) \vee t$.

Figure 3 shows the most important parts of the schema syntax (eliding some details, such as defaults, integers, and array schemas, that are not needed for examples in this paper). The start symbol, *schema*, can be an enumeration, a range, an object, or it can nest other schemas by using one of three logic connectives. An *enum* schema specifies a categorical hyperparameter with a choice between one or more values. Figure 4 includes multiple examples of enums: $[\mathit{mle}]$, $[\mathit{true}, \mathit{false}]$, $[0.25]$, $[\mathit{linear}, \mathit{sag}, \mathit{lbfgs}]$, etc. A *range* schema specifies a continuous hyperparameter with a minimum and maximum (most AutoML optimizers do best with bounded ranges) and an optional distribution *distr*. Figure 4 includes two examples of ranges: $(0..1)$ and $(0..0.5)$. An *object* schema specifies a dictionary of zero or more hyperparameter schemas. For example, Figure 4 specifies one hyperparameter for the PCA operator from sklearn [6] and two each for J48 from Weka [14] and for LR from sklearn.

The schema syntax supports three logic connectives disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$). Disjunction describes a union type; for example, in Figure 4, the N hyperparameter of PCA can specify the number of components with either a range $(0..1)$ or an enum $[mle]$. Conjunction describes an intersection type and negation can be used to describe an implication. These connectives help express *constraints* for ruling out combinations of hyperparameters that are individually valid but collectively invalid. For example, with Weka's J48 operator, if R is *true* (use reduced error pruning), then C must be 0.25 (cannot set the pruning confidence to other values). Figure 4 specifies this via $\Rightarrow$, which is syntactic sugar for the basic logic connectives. Hyperparameter constraints are useful for pruning invalid areas from the search space, so AutoML search can focus on valid areas instead.

To make schemas easier to use, we decided not to invent our own new schema language, but instead to adopt a mature existing language with a broad user base and tooling ecosystem. One choice we considered was Python 3 types [32]. Unfortunately, Python 3 types are not expressive enough for specifying constraints. But fortunately, there is another language that also works well with Python and is used extensively by a large community well beyond that of AutoML: JSON Schema [26]. Most constructs from Figure 3 have exact equivalents in JSON Schema. For instance, a range $(a..b)$ can be written as `{"type": "number", "minimum":a, "maximum":b}`. Lale adds a few extensions to JSON Schema. For instance, it supports an optional `"distribution"` keyword in a `"number"` schema; and it supports an `"operator"` type for hyperparameters that accept a nested machine-learning operator.

3 Gradual Automation

The previous section described pipeline combinators and hyperparameter schemas. This section builds on that foundation to describe a programming model for gradual AutoML. Here, *gradual* means that users get to use total automation if they so desire, but can also exercise fine-grained manual control over any aspects of the pipeline. The programming model design was guided by three principles. First, *progressive disclosure* lets users start by learning just a subset of the constructs, only requiring them to learn more constructs as needed to accomplish more tasks. Second, *orthogonality* minimizes the number of independent constructs to make the programming model easy to learn, while maximizing the ways these constructs can be plugged together to make the programming model expressive [20]. Third, the *principle of least surprise* makes each construct look and behave the way most users would expect. That is why we started from well-established programming models sklearn [6] and JSON Schema [26] and try to be faithful to their expectations.

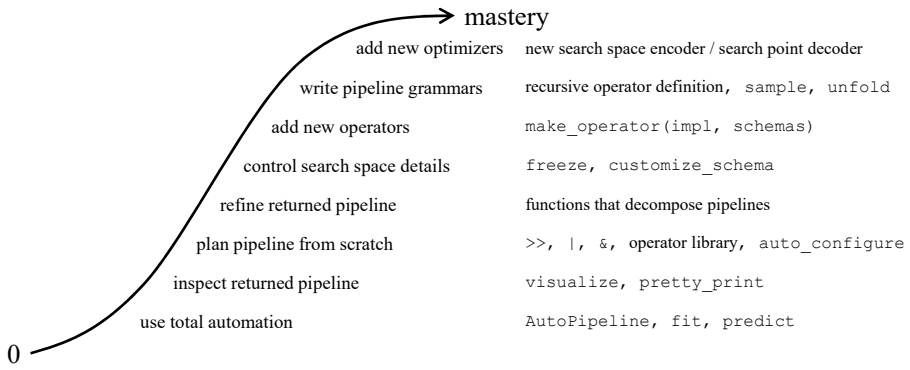

Figure 5: Gradual automation with progressive disclosure.

Figure 5 show the learning curve of gradual AutoML from zero to mastery. The rest of this section describes each set of constructs, starting at the bottom and working upwards one layer at a time.

Use total automation. Lale provides an `AutoPipeline` operator. From the outside, it looks and feels just like any sklearn operator with `fit` and `predict` methods. Internally, `fit` runs AutoML on a predefined planned pipeline, which users need not dwell on. Hence, total automation with Lale is trivial to learn for users who already know sklearn (least surprise).

Inspect returned pipeline. One level up from total automation is to look at its result: even users who have little interest in controlling the pipeline may want to see it for transparency. Lale provides a `visualize` method and a round-trippable `pretty_print` method that returns a pipeline's Python code.

Plan pipeline from scratch. Learning about the combinators and the `auto_configure` method from Section 2 brings users to the next level. To further assist in using Lale's collection of 216 operators, Lale automatically generates documentation from schemas and uses schemas for error checking.

Refine returned pipeline. Data science, even with automation, is often iterative. After an AutoML search (whether with full automation or with a hand-planned pipeline), users may want to tweak the pipeline for another search. Lale supports this with functions for breaking down a pipeline into smaller pieces, which can then be composed back into a new pipeline using combinators (orthogonality).

Control search space details. To exercise even more control in a localized and composable way, users can disable AutoML search for a sub-pipeline with `freeze_trainable`, or they can disable refitting with `freeze_trained`. Furthermore, users may want to locally tweak the schema for some hyperparameters of an operator, such as adjusting a range. Given an operator `A`, calling `A.customize_schema(..)` returns a new operator that is the same as `A` except for the given schema tweaks. Again, these construct are orthogonal to combinators: frozen or customized operators can be used like any others.

Add new operators. Given an operator without a schema, Lale infers a simple schema that just sets all hyperparameters to their defaults. Users can then call `customize_schema` to make them selectively tunable. Or they can write a comprehensive schema and attach it to the operator using `make_operator`.

Write pipeline grammars. Some AutoML tools [22, 8, 10] explore not a rigid template for pipeline topologies but a possibly unbounded space of topologies. Lale lets users define such search spaces by declaring *recursive operators*, which are self-referential definitions (e.g., `g.p = g.p >> q`). Recursive operators are orthogonal to the other constructs such as combinators, with which they can nest. The methods `sample` and `unfold` turn these into normal Lale pipelines suitable for `auto_configure`.

Add new optimizers. Only few users get so close to mastery that they desire to add their own optimizers. While the details will vary on a case-by-case basis, the next section describes a principled translation scheme that helped Lale target several existing optimizers.

4 Translation Scheme

This paper adds a third execution mode to sklearn-style pipelines: the first two modes are predict-or-transform and fit, and the third mode is AutoML search. Lale's API for this mode is `auto_configure`, which first translates the pipeline to a search space for a given optimizer and then runs that optimizer.

Figure 6 details that interaction to illustrate how the translation scheme (encode and decode) interacts with an optimizer and how it uses the first two execution modes (predict and fit). The starting point is a *planned* pipeline, which has operators whose hyperparameters need to be tuned or operator choices (`|` combinator) whose operators need to be selected. Step "encode search space" translates the planned pipeline to an optimizer-specific search space. Once called, the optimizer loops over trials, which are search points it selects based on its solver, e.g., TPE [4]. Step "decode search point" turns an optimizer-specific *search point* (i.e., a specific value combination in the search space) into the configuration needed for instantiating a trainable pipeline. In a *trainable* pipeline, all hyperparameters and operator choices are bound to specific values. Fitting the trainable pipeline to training data (folds) yields a *trained* pipeline, where the learnable coefficients of all operators are also bound. The next step is to call predict on the trained pipeline and to score the result to compute a loss for the optimizer. After wrapping up its loop, the optimizer returns the best trial, and `auto_configure` returns the corresponding trained pipeline.

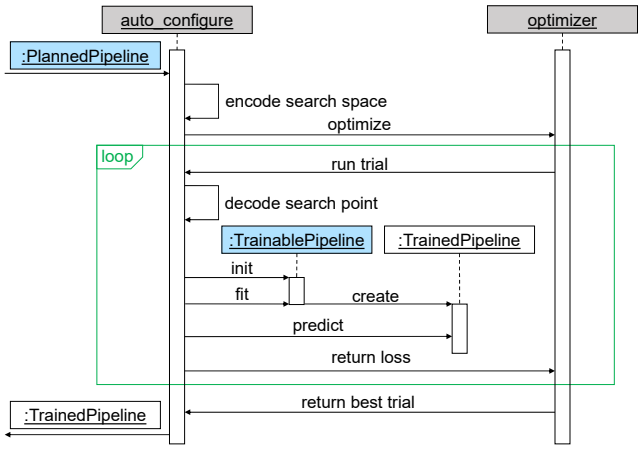

Figure 6: Interaction with an optimizer.

The translation scheme is one of the contributions of this paper. Its job is to enable the modular combinator-based syntax for users, by weaving together the operators and their hyperparameter schemas and rewriting them to a form suitable for a given optimizer. Users can thus make local changes using orthogonal constructs without worrying about their impact on the rest of the pipeline. To target a variety of optimizers, we structured the translation scheme as a series of rewriting *passes* between different *forms*. Each form has its own syntax restricting how a search space can be expressed. Figure 3 shows the starting form. Each pass is designed to be semantics-preserving, in the sense that the search space encodes the same set of pipeline instances before and after the pass. The running example for this section is the pipeline `PCA >> (J48 | LR)`; Figure 4 shows its schemas in starting form.

Normalize. Figure 7 shows the normal form, which is a disjunction of objects. This form is crafted to simplify later rewriting passes by letting them start from a less feature-laden form. While nesting is still possible in normal form, it is restricted to objects. Also, once in normal form, schemas contain no conjunctions or negations, which would inhibit local reasoning. Normalization works in a bottom-up pass. At each level, it performs disjunction hoisting using rewrites such as De-Morgan's law. Negation and conjunction are pushed into primitive schemas. This can temporarily give rise to uninhabited or trivial nested schemas, which the pass simplifies away with another set of rewrites. Our translation scheme normalizes the schema of each operator in a pipeline separately.

```
schema ::= object (∨ object)*
object ::= dict { (NAME : prop)* }
prop   ::= enum | range | object
enum   ::= [ VALUE+ ]
range  ::= ( NUM .. NUM ) distr?
distr  ::= "uniform" | "loguniform"
```

$$
\begin{aligned}
\mathit{PCA} &: \text{dict}\{N\colon (0..1)\} \vee \text{dict}\{N\colon [\mathit{mle}]\} \\
\mathit{J48} &: \text{dict}\{R\colon [\mathit{false}], C\colon (0..0.5)\} \vee \\
&\quad\ \text{dict}\{R\colon [\mathit{true}, \mathit{false}], C\colon [0.25]\} \\
\mathit{LR} &: \text{dict}\{S\colon [\mathit{linear}], P\colon [\mathit{l1}, \mathit{l2}]\} \vee \\
&\quad\ \text{dict}\{S\colon [\mathit{linear}, \mathit{sag}, \mathit{lbfgs}], P\colon [\mathit{l2}]\}
\end{aligned}
$$

Figure 7: Normalized schemas: syntax (left) and example (right).

Combine. Figure 8 shows the combined form, which represents the entire pipeline together instead of each operator separately. Combining encodes combinators as follows. The `>>` and `&` combinators turn into an object schema (dict), where each operator becomes a step s_i. The `|` combinator turns into a disjunction schema ($\vee$), with an additional discriminant D to track what was chosen. Higher-order operators yield nested object schemas (dict). The encoding preserves sufficient information to enable later decoding. After this pass, the encoding is conceptually similar to that of Hyperopt [5], so our implementation provides a simple backend translating it into a Hyperopt search space.

```
space  ::= object
object ::= dict { (NAME : prop)* }
prop   ::= enum | range | disj
disj   ::= object (∨ object)*
```

$$
\text{dict}\left\{\begin{array}{l}
s_0\colon \text{dict}\{N\colon (0..1)\} \vee \text{dict}\{N\colon [\mathit{mle}]\} \\
s_1\colon \left(\begin{array}{l}
\left(\begin{array}{l}\text{dict}\{D\colon [\mathit{J48}], R\colon [\mathit{false}], C\colon (0..0.5)\} \vee \\ \text{dict}\{D\colon [\mathit{J48}], R\colon [\mathit{true}, \mathit{false}], C\colon [0.25]\}\end{array}\right) \vee \\
\left(\begin{array}{l}\text{dict}\{D\colon [\mathit{LR}], S\colon [\mathit{linear}], P\colon [\mathit{l1}, \mathit{l2}]\} \vee \\ \text{dict}\{D\colon [\mathit{LR}], S\colon [\mathit{linear}, \mathit{sag}, \mathit{lbfgs}], P\colon [\mathit{l2}]\}\end{array}\right)
\end{array}\right)
\end{array}\right\}
$$

Figure 8: Combined search space: syntax (left) and example (right).

Flatten. Most other optimizers require a less hierarchical search space than the combined form. Figure 9 shows the flattened form, which is a disjunction of objects without further nesting. The flatten pass hoists properties from nested objects, renaming them as needed for uniqueness (in a way that enables later decoding). The flattened form is close to what SMAC [13] needs, except that objects of different disjuncts have different properties. In SMAC, all disjuncts have the same properties, with conditionals on the side indicating which variables are relevant. The SMAC backend thus generates these conditionals, and renders the space in SMAC's parameter configuration space format [29].

The ADMM backend uses a partially flattened form. ADMM [19] supports a pipeline of operator choices. The backend thus leaves the top-level spine of a pipeline intact as a pipeline of choices. If one of the choices contains a complex operator (e.g. a nested pipeline), it is flattened as for SMAC.

Discretize. The optimizers GridSearchCV and HalvingGridSearchCV from sklearn [6] require continuous ranges to be made categorical. Discretization for a hyperparameter first includes the default and then samples a user-configurable number of additional values from the range based on the prior distribution. Figure 10 shows the discretized form when generating two values per range.

$$
\begin{aligned}
\textit{space} &::= \textit{object}\ (\vee\ \textit{object})^{*} \\
\textit{object} &::= \texttt{dict}\ \{\ (\textit{NAME} : \textit{prop})^{*}\ \} \\
\textit{prop} &::= \textit{enum} \mid \textit{range}
\end{aligned}
\qquad
\begin{aligned}
&\ \ \text{dict}\{N\colon (0..1), D\colon [\textit{J48}], R\colon [\textit{false}], && C\colon (0..0.5)\} \\
&\vee \text{dict}\{N\colon (0..1), D\colon [\textit{J48}], R\colon [\textit{true}, \textit{false}], && C\colon [0.25]\ \} \\
&\vee \text{dict}\{N\colon [\textit{mle}],\ D\colon [\textit{J48}], R\colon [\textit{false}], && C\colon (0..0.5)\} \\
&\vee \text{dict}\{N\colon [\textit{mle}],\ D\colon [\textit{J48}], R\colon [\textit{true}, \textit{false}], && C\colon [0.25]\ \} \\
&\vee \text{dict}\{N\colon (0..1), D\colon [\textit{LR}],\ S\colon [\textit{linear}], && P\colon [\textit{l1}, \textit{l2}]\ \} \\
&\vee \text{dict}\{N\colon (0..1), D\colon [\textit{LR}],\ S\colon [\textit{linear}, \textit{sag}, \textit{lbfgs}], && P\colon [\textit{l2}]\ \} \\
&\vee \text{dict}\{N\colon [\textit{mle}],\ D\colon [\textit{LR}],\ S\colon [\textit{linear}], && P\colon [\textit{l1}, \textit{l2}]\ \} \\
&\vee \text{dict}\{N\colon [\textit{mle}],\ D\colon [\textit{LR}],\ S\colon [\textit{linear}, \textit{sag}, \textit{lbfgs}], && P\colon [\textit{l2}]\ \}
\end{aligned}
$$

Figure 9: Flattened search space: syntax (left) and example (right).

$$
\begin{aligned}
\textit{space} &::= \textit{object}\ (\vee\ \textit{object})^{*} \\
\textit{object} &::= \texttt{dict\{}\ (\textit{NAME} : \textit{enum})^{*}\ \}
\end{aligned}
\qquad
\begin{aligned}
&\ \ \text{dict}\{N\colon [0.50, 0.01], D\colon [\textit{J48}], R\colon [\textit{false}], C\colon [0.25, 0.01]\} \\
&\vee \text{dict}\{N\colon [0.50, 0.01], D\colon [\textit{J48}], R\colon [\textit{true}, \textit{false}], C\colon [0.25]\} \\
&\vee \text{dict}\{N\colon [\textit{mle}], D\colon [\textit{J48}], R\colon [\textit{false}], C\colon [0.25, 0.01]\} \\
&\vee \ldots \\
&\vee \text{dict}\{N\colon [\textit{mle}], D\colon [\textit{LR}], S\colon [\textit{linear}, \textit{sag}, \textit{lbfgs}], P\colon [\textit{l2}]\}
\end{aligned}
$$

Figure 10: Discretized search space: syntax (left) and example (right).

5 Results

RQ1: How usable is gradual AutoML with combinators? To explore this research question, we compare Lale to the popular sklearn machine learning framework in a user study with 18 participants. The study design is between-subjects [7] where participants are randomly assigned to a Lale or sklearn version of the study. We asked participants to perform four tasks involving sample machine learning pipelines: (T1) inspecting and understanding an existing pipeline, (T2) refining the given pipeline by replacing operators and hyperparameters, (T3) debugging a failing pipeline, and (T4) searching across a space of potential pipelines. The tasks are based on the principle of *progressive disclosure* in that we provide a sample pipeline (Figure 11) using common manual constructs in either sklearn or Lale format and ask participants to perform tasks that use a progressively expanding set of constructs. All participants are experienced in using sklearn, come from the same large technology company, and were not compensated for their voluntary participation. The supplemental material describes the detailed study design and methodology, including the sklearn and Lale notebooks used.

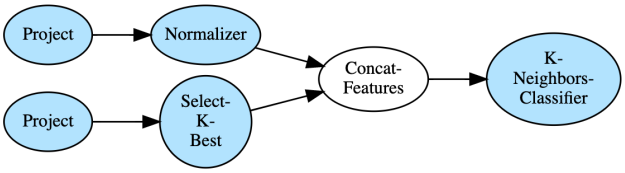

Figure 11: Sample pipeline used in user studies.

One surprising result of the user study is that four (44%) of the sklearn participants were not able to correctly identify the branching structure of the preprocessing subpipelines of the sample pipeline during T1 (seen in Figure11), compared to a single Lale participant (11%). Two of the Lale participants also used inspection features in Lale such as the `visualize` and `pretty_print` methods. In T4, two (22%) of the sklearn participants were either unable to correctly (or at all) implement some method of searching across a space of potential pipelines whereas all of the Lale participants were able to complete the task. When implementing T4, most of the Lale participants (6 or 67%) used the or combinator (`|`) and the AutoML execution mode to perform automated search as opposed to more manual methods. This is compared to two (22%) sklearn participants using automated search via GridSearchCV. The relatively high success rate of Lale participants in both understanding existing pipelines that use the and combinator (`&`) and to compose new planned pipelines that use the or combinator (`|`) suggests that new Lale users are able to pick up and use these independent constructs. This supports the principle of *orthogonality* in the Lale programming model. Automated search also tends to be more concise than manual implementations for the search task, perhaps contributing to Lale participants on average implementing the task in 25 fewer lines of code (LoC) (59% less) compared to sklearn participants. Lale participants also on average performed all of the tasks faster than sklearn participants, over 2.5 minutes faster total (13% faster). (In reality, the time difference would likely be even higher, since we stopped participants who take a manual approach early, see supplement for details.) For T2 and T3, participants performed equally well in both versions of the study. We note that some Lale participants where able to refine pipelines without referring to documentation, relying on their sklearn knowledge to replace operators and define their

hyperparameters. This supports the *principle of least surprise* in that Lale operators work in ways that experienced sklearn users expect. Table 1 summarizes the results from the user study.

Table 1: Summary of user study results.

VERSION	COUNT	T1 CORRECT	T4 CORRECT	T4 LOC MEDIAN (ALL)	TOTAL TIME MEAN (STDDEV)
Lale	9	89%	100%	10 (7, 8, 9, 9, 10, 10, 10, 14, 75)	17:36 (5:03)
Sklearn	9	56%	78%	23 (12, 15, 16, 22, 24, 43, 100, 100)	19:26 (6:34)
Total	18	72%	89%	14	18:54 (5:45)

RQ2: Does the translation scheme work for diverse planned pipelines? To explore this research question, we used Lale's `auto_configure` on several planned pipelines with Hyperopt and its TPE solver, on 14 datasets from OpenML [33] (CC-BY license). Table 2 summarizes the results. Column AUTOSKL uses auto-sklearn [13] as a state-of-the-art baseline; LALE-AUTO uses Lale's `AutoPipeline` operator from Section 3; LALE-AD3M and LALE-TPOT use Lale grammars inspired by AlphaD3M [10] and TPOT [22] unfolded with a maximal depth of 3; LALE-ADB uses a pipeline with a choice of preprocessors and the `AdaBoostClassifier` higher-order operator; and ASKL-ADB is auto-sklearn [13] with the set of classifiers limited to AdaBoost with only preprocessing. (See supplement for details.) We used a 66:33% train:test split with 5-fold cross validation on the train set during optimization. We used a 2.0GHz virtual machine with 32 cores and 128GB memory and gave each search a 1 hour time budget with a timeout of 6 minutes per trial, which corresponds to the default setting of auto-sklearn. All searches succeed and most yield performance in the same ballpark as auto-sklearn (aside from shuttle, which triggers an auto-sklearn issue that other users have also encountered).

Table 2: Test accuracy for 14 OpenML classification tasks, using 4 Lale search spaces with Hyperopt.

	Absolute accuracy mean (and stddev) over 5 runs						$100 * (\text{LALE}/\text{AUTOSKL} - 1)$			
DATASET	AUTOSKL	LALE-AUTO	LALE-TPOT	LALE-AD3M	LALE-ADB	ASKL-ADB	AUTO	TPOT	AD3M	ADB
australian	85.1(0.4)	86.2 (0.0)	85.9 (0.6)	86.8 (0.0)	86.0 (1.6)	84.7 (3.1)	1.3	0.9	2.0	1.1
blood	77.9(1.4)	75.3 (0.0)	77.5 (2.5)	74.7 (0.7)	77.1 (0.7)	74.7 (0.8)	-3.3	-0.5	-4.0	-1.0
breast-cancer	73.0(0.6)	73.0 (0.0)	71.4 (1.1)	69.5 (3.3)	70.9 (2.0)	72.4 (0.5)	0.0	-2.3	-4.9	-2.9
car	99.4(0.1)	97.7 (0.0)	99.1 (0.1)	92.7 (0.6)	98.3 (0.3)	98.2 (0.2)	-1.6	-0.2	-6.7	-1.1
credit-g	76.6(1.2)	75.7 (0.0)	74.1 (0.5)	74.8 (0.4)	76.1 (1.3)	76.2 (1.0)	-1.1	-3.2	-2.4	-0.7
diabetes	77.0(1.3)	76.3 (0.0)	76.4 (1.1)	77.9 (0.2)	76.0 (0.5)	75.0 (1.0)	-0.9	-0.8	1.1	-1.3
jungle-chess	88.1(0.2)	92.4 (0.0)	88.9 (2.0)	74.1 (2.0)	89.4 (2.3)	86.9 (0.2)	4.9	0.9	-15.8	1.5
kc1	83.8(0.3)	83.4 (0.0)	83.5 (0.5)	83.6 (0.2)	83.3 (0.4)	84.0 (0.3)	-0.5	-0.4	-0.2	-0.6
kr-vs-kp	99.7(0.0)	99.5 (0.0)	99.4 (0.0)	96.8 (0.1)	99.5 (0.1)	99.5 (0.2)	-0.2	-0.3	-2.9	-0.2
mfeat-factors	98.7(0.1)	97.1 (0.0)	97.9 (0.5)	97.5 (0.1)	97.5 (0.4)	97.9 (0.1)	-1.6	-1.5	-1.2	-1.2
phoneme	90.3(0.4)	89.5 (0.0)	89.6 (0.4)	76.6 (0.0)	90.1 (0.4)	91.4 (0.2)	-0.8	-0.8	-15.2	-0.2
shuttle	87.3(11.6)	100.0 (0.0)	99.9 (0.0)	99.9 (0.0)	100.0 (0.0)	100.0 (0.0)	14.5	14.5	14.4	14.6
spectf	87.9(0.9)	87.7 (0.0)	88.4 (2.2)	83.6 (6.9)	88.4 (2.6)	89.7 (2.9)	-0.2	0.6	-4.9	0.6
sylvine	95.4(0.2)	95.0 (0.0)	94.4 (0.7)	91.3 (0.1)	95.1 (0.2)	95.1 (0.1)	-0.4	-1.1	-4.3	-0.3

RQ3: Does the translation scheme work for diverse optimizer backends? To explore this research question, Table 3 lists test accuracies using 9 different optimizers for the OpenML phoneme dataset. The planned pipeline uses a scaler, a polynomial feature generator, and a choice between XGBoostClassifier and LGBMClassifier. The other experimental settings are the same as above. All translations succeed, and most yield reasonable accuracy, except for the ones that timed out.

Table 3: Test accuracy (mean and stddev) for the phoneme dataset using multiple optimizer backends.

Hyperopt			Hyperband	ADMM			GridSearchCV	Halving-
TPE	Anneal	Rand		RND	BOBa	GPRND		GridSearchCV
88.75 (0.2)	88.94 (0.3)	88.75 (0.5)	87.94 (0.2)	89.00 (0.2)	89.14 (0.2)	88.61 (0.3)	Timed out	Timed out

6 Related Work

Combinators, which were discovered in the 1920s [28], have long been prominent in functional programming languages such as Lisp [30]. The advantage of their tacit style has been recognized more broadly; for instance, the Unix pipe [21] can be viewed as a combinator. Combinators have also enabled some sophisticated innovations such as parser combinators [17].

Combinators and data science. MapReduce [9] is a successful programming model for large-scale data processing named after two combinators [16]. Allison [2] describes Haskell combinators for

converting between different machine-learning model types, but does not discuss AutoML. Sklearn [6] provides combinators (`make_pipeline` and `make_union`) as well as some basic AutoML (`GridSearchCV`, `HalvingGridSearchCV`, and `RandomizedSearchCV`) but only integrates combinators with AutoML loosely, instead requiring users to perform manual name-mangling and non-compositional repetition. Lale builds on sklearn, extending and streamlining its combinators and tightly integrating them with AutoML. Pilat et al. [27] describe a combinator language suitable as a target for automatic code generation with genetic programming, but not intended to be written by hand. Unlike our work, they focus on total (not gradual) automation and do not provide an or combinator (`|`).

AutoML programming models. Auto-WEKA demonstrated how to support conditional hyperparameters on multiple solvers, but the paper focused on a single planned pipeline, not gradual automation [31]. While both hyperopt-sklearn [15] and auto-sklearn [13] support some gradual automation, instead of tightly integrating gradual automation with combinators, they require users to drop to a bespoke lower-level API: Hyperopt for hyperopt-sklearn and SMAC's parameter configuration space format [29] for auto-sklearn. We argue that JSON Schema [26], being a language with a track record of wide-spread use by programmers who are not AutoML experts, is less low-level; e.g., it is the backbone of OpenAPI [23]. In Optuna, users write imperative code that first samples values, then uses either if-statements for operator choice or passes values to constructors for hyperparameter tuning [1]. In contrast, combinators are more concise, more localized, and less surprising to sklearn users. In PyGlove, users start from a manually written neural network which they then "hyperify" by replacing a component or hyperparameter with a "hyper value" [25]. Their notion of "hyper" is similar to our notion of "planned", but they do not use combinators, and they focus only on neural architecture search. AutoGluon-Tabular focuses on total automation and the paper does not discuss gradual automation [11]. In AutoGOAL, users specify a hierarchy of pre-defined "indicator" types, then the system tries out any pipeline of operators that gets from the input to the output [12]. AutoGOAL's API deviates from sklearn conventions, and while it uses the syntax of Python 3 types, it does not capture constraints, nor is it compatible with Python 3 type checkers.

7 Limitations and Societal Impacts

A limitation of the user study is that it used only 18 participants (9 per version). Another potential limitation is that researcher bias may be present in the design of the user study tasks. We designed the tasks to showcase and validate aspects of the Lale programming model which may potentially bias tasks in favor of Lale. We attempt to mitigate this by only selecting participants who are experienced in sklearn and by only allowing Lale newcomers to participate in the Lale version of the study. One limitation of the translation scheme is that it is not guaranteed to handle all JSON Schema features; in fact, we encountered and fixed some limitations early during development. Fortunately, there were no new limitations recently, and there is theoretical work that could be used to tackle them if they arise [3]. A limitation of schema-based constraints is that they currently do not capture interdependence between operators. We mitigate that by providing not just hyperparameter schemas but also dataset schemas. A limitation of the programming model is that, while it achieves a high level of sklearn compatibility, Python code that explicitly reflects over dynamic types can still expose differences. We mitigate this by testing that Lale interoperates well with a broad set of sklearn features including metrics, cross-validation, operators, etc.

One potential societal impact is that enabling automation of searching machine learning pipelines may encourage computationally-heavy approaches to data science tasks and raise CO_2 emissions. On the other hand, we hope gradual automation will encourage data scientists to use their domain knowledge to more selectively and efficiently automate their machine learning work.

8 Conclusion

This paper presents combinators for gradual AutoML, along with a novel translation scheme from combinators to existing optimizers. We have implemented these concepts in an open-source library called Lale, while ensuring they interoperates smoothly with the sklearn ecosystem. Besides being open-source, Lale is also actively used in a commercial AutoML product by a leading techology company, highlighting the real-world value of the underlying concepts. Our ultimate goal is to make AutoML as accessible as manual ML by offering a gradual combination of both.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Conference on Knowledge Discovery and Data Mining (KDD) – Applied Data Science Track*, pages 2623–2631, 2019.

[2] Lloyd Allison. Models for machine learning and data mining in functional programming. *Journal on Functional Programming (JFP)*, 15(1):15–32, January 2005.

[3] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Not elimination and witness generation for JSON Schema. In *Conférence sur la Gestion de Données (BDA)*, October 2020.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Conference on Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[5] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D. Cox. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1), 2015.

[6] 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.

[7] John W. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE publications, 4th edition, 2013.

[8] Alex G. C. de Sá, Walter José G. S. Pinto, Luiz Otavio V. B. Oliveira, and Gisele L. Pappa. RECIPE: A grammar-based framework for automatically evolving classification pipelines. In *European Conference on Genetic Programming (EuroGP)*, pages 246–261, 2017.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[10] Iddo Drori, Yamuna Krishnamurthy, Raoni Lourenco, Remi Rampin, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Automatic machine learning by pipeline synthesis using model-based reinforcement learning and a grammar. In *Workshop on Automatic Machine Learning (AutoML)*, 2019.

[11] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhangh, Pedro Larroy, Mu Li, and Alexander Smola. AutoGluon-Tabular: Robust and accurate AutoML for structured data. In *ICML Workshop on Automated Machine Learning (AutoML@ICML)*, July 2020.

[12] Suilan Estevez-Velarde, Alejandro Piad-Morffis, Yoan Gutiérrez, Andrés Montoyo, Rafael Munoz, and Yudivian Almeida-Cruz. Solving heterogeneous automl problems with AutoGOAL. In *ICML Workshop on Automated Machine Learning (AutoML@ICML)*, 2020.

[13] 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.

[14] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter*, 11(1):10–18, November 2009.

[15] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: Automatic hyperpa-rameter configuration for scikit-learn. In *Python in Science Conference (SciPy)*, pages 32–37, 2014.

[16] Ralf Lämmel. Google's MapReduce programming model — revisited. *Science of Computer Programming*, 70(1):1–30, 2007.

[17] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2002.

[18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research (JMLR)*, 18(185):1–52, 2018.

[19] Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, Djallel Bouneffouf, Gregory Bramble, Horst Samulowitz, Dakuo Wang, Andrew Conn, and Alexander G. Gray. An ADMM based framework for AutoML pipeline configuration. In *Conference on Artificial Intelligence (AAAI)*, pages 4892–4899, 2020.

[20] B.J. Mailloux, J.E.L. Peck, and C.H.A. Koster. *Report on the Algorithmic Language ALGOL 68*. Springer, 1969.

[21] M. Douglas McIlroy. A research UNIX reader: Annotated excerpts from the programmer's manual, 1971–1986. Technical report, Bell Labs, 1987. https://www.cs.dartmouth.edu/~doug/reader.pdf.

[22] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. Automating biomedical data science through tree-based pipeline optimization. In *European Conference on the Applications of Evolutionary Computation (EvoApplications)*, pages 123–137, 2016.

[23] OpenAPI Initiative. OpenAPI specification (fka swagger restful api documentation specification), 2014. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md.

[24] David L. Parnas, John E. Shore, and David Weiss. Abstract types defined as classes of variables. In *Conference on Data: Abstraction, Definition and Structure*, pages 149–154, 1976.

[25] Daiyi Peng, Xuanyi Dong, Esteban Real, Mingxing Tan, Yifeng Lu, Gabriel Bender, Hanxiao Liu, Adam Kraft, Chen Liang, and Quoc Le. PyGlove: Symbolic programming for automated machine learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[26] 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.

[27] Martin Pilat, Tomas Kren, and Roman Neruda. Asynchronous evolution of data mining workflow schemes by strongly typed genetic programming. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 577–584, 2016.

[28] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92(3):305–316, 1924.

[29] SMAC Parameter Configuration Space (PCS). https://automl.github.io/SMAC3/master/options.html#parameter-configuration-space-pcs.

[30] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[31] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 847–855, 2013.

[32] Guido van Rossum, Jukka Lehtosalo, and Lukasz Langa. PEP 484 – Type Hints, 2014. https://www.python.org/dev/peps/pep-0484/.

[33] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations Newsletter*, 15(2):49–60, June 2014.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 7.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 7.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 5.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Section 5, which reports standard deviations after running experiments 5 times.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See supplemental material.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See Section 5.
   (b) Did you mention the license of the assets? [Yes] See Section 5.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] The supplemental material includes a copy of the code of the Lale library.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We used open datasets and open-source software.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We used standard datasets from OpenML [33] that contained neither personally identifiable information nor offensive content.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes] See supplemental material.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] There were no risks to participants.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] Participants were volunteers and did not get compensated for joining our study.