LALE: Consistent Automated Machine Learning

Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar
IBM Research, USA

ABSTRACT
Automated machine learning makes it easier for data scientists to develop pipelines by searching over possible choices for hyperparameters, algorithms, and even pipeline topologies. Unfortunately, the syntax for automated machine learning tools is inconsistent with manual machine learning, with each other, and with error checks. Furthermore, few tools support advanced features such as topology search or higher-order operators. This paper introduces Lale, a library of high-level Python interfaces that simplifies and unifies automated machine learning in a consistent way.

1 INTRODUCTION
Machine learning (ML) is widely used in various data science problems. There are many ML operators for data preprocessing (scaling, missing imputation, categorical encoding), feature extraction (principal component analysis, non-negative matrix factorization), and modeling (boosted trees, neural networks). A machine learning pipeline consists of one or more operators that take the input data through a series of transformations to finally generate predictions. Given the plethora of ML operators (for example, in the widely used scikit-learn library [5]), the task of finding a good ML pipeline for the data at hand (which involves not only selecting operators but also appropriately configuring their hyperparameters) can be tedious and time consuming if done manually. This has led to a wider adoption of automated machine learning (AutoML), with the development of novel algorithms (such as SMAC [12], hyperopt [3], and subsequent recent work [16]), open source libraries (auto-sklearn [10], hyperopt-sklearn [4, 15], TPOT [17]), and even commercial tools.

Scikit-learn provides a consistent programming model for manual ML [5], and various other tools (such as XGBoost [6], LightGBM [14], and Tensorflow's tf.keras.wrappers.scikit_learn) maintain consistency with this model whenever appropriate. AutoML tools usually feature two pieces – (i) a way to define the search space corresponding to the pipeline topology, the operator choices in that topology, and their respective possible hyperparameter configurations, and (ii) an algorithm that explores this search space to optimize for predictive performance. Many of these also try to maintain consistency with the scikit-learn model but only when the user gives up all control and lets the tool completely automate the ML pipeline configuration, eschewing item (i) above. Users often want to retain some control over the automation, for example, to comply with domain-specific requirements by specifying operators or hyperparameter ranges to search over. While fine-grained control over the automation is possible (we will discuss examples), users need to manually configure the search space. This search space specification differs for each of the different AutoML tools and differs from the manual programming model.

We believe that proper abstractions are necessary to provide a consistent programming model across the entire spectrum of controlled automation. To this end, we introduce Lale, an open-source Python library, built upon mathematically grounded abstractions.

Figure 1: LALE example for consistent automated machine learning, explained in Section 3.
of the elements of the pipeline (for example, topology, operator, hyperparameter configurations), and designed around scikit-learn [5] and JSON Schema [18]. Lale provides a programming interface for specifying pipelines and search spaces in a consistent manner while providing fine-grained control across the automation spectrum. The abstractions also allow us to provide a consistent programming interface for capabilities for which no such interface currently exists: (i) search spaces with higher-order operators (operators, such as ensembling, that have other operators as hyperparameters), and (ii) search spaces that include search for the pipeline topology via context-free grammars [7]. The contributions of this paper are:

1. A grammar syntax for pipeline topology search that is a natural extension of the pipeline syntax.
2. A programming model for pipelines and search spaces in a consistent manner while providing fine-grained control across the automation spectrum.
3. Higher-order operators (for ensembles, batching, etc.) with an automatic search space generation for nested pipelines.
4. A grammar syntax for pipeline topology search that is a natural extension of the pipeline syntax.

Overall, we hope LALe will make data scientists more productive at finding pipelines that are consistent with their requirements and yield high predictive performance.

2 PROBLEM STATEMENT
Consistency is a core problem for AutoML and existing libraries fall short on this front. This section uses concrete examples from popular AutoML libraries to present four shortcomings in existing systems. We strive to do so in a factual and constructive way.

P1: Provide a consistent programming model across different AutoML tools. Compared to GridSearchCV, Bayesian optimizers such as hyperopt-sklearn [15] and auto-sklearn [10] speed up search using smarter search strategies. Unfortunately, each of these AutoML tools comes with its own syntax and concepts.

Figure 2 shows an example from the no-automation end of the spectrum using scikit-learn [5]. The code assembles a two-step pipeline of a PCA transformer and a LinearSVC classifier and manually sets their hyperparameters, for example, n_components=4.

The example in Figure 3 automates hyperparameter tuning and operator selection using GridSearchCV from scikit-learn. Lines 1–3 resemble Figure 2. Lines 4–11 specify a search space, consisting of a list of two dictionaries. In the first dictionary, Line 7 specifies the list of values to search over for the n_components hyperparameter of the PCA operator; Line 8 specifies the classify step of the pipeline to be a LinearSVC operator; and Line 9 specifies the list of values to search over for the C hyperparameter of the LinearSVC operator. The second dictionary is similar, but specifies the RandomForestClassifier.

The syntax for a pipeline (Figure 2 Lines 1–3) differs from that for a search space (Figure 3 Lines 4–11). The mental model is that operators and hyperparameters are pre-specified and then the search space selectively overwrites them with different choices. To do so, the code uses strings to name steps and hyperparameters, with a double underscore (_ _) name mangling convention to connect the two. Relying on strings for names can cause hard-to-detect mistakes [1]. In contrast, using a single syntax for both manual and automated pipelines would make them more consistent and would obviate the need for mangled strings to link between the two.

P2: Provide a consistent programming model across different AutoML tools. Compared to GridSearchCV, Bayesian optimizers such as hyperopt-sklearn [15] and auto-sklearn [10] speed up search using smarter search strategies. Unfortunately, each of these AutoML tools comes with its own syntax and concepts.

Figure 4 shows an example using the hyperopt-sklearn [15] wrapper for hyperopt [3]. Line 1 specifies a discrete search space N with a logarithmic prior, a range from 2..8, and a quantization to multiples of 1. Line 2 specifies a continuous search space C with a logarithmic prior and a range from 1..1000. Line 4 sets the transform step of the pipeline to pca with hyperparameter n_components=N. Lines 5–7 set the classify step to a choice between linear_svc and random_forest.
Figure 5 shows the same example using auto-sklearn [10]. While power users can use the ConfigurationSpace used by SMAC [12] to adjust search spaces for individual hyperparameters, we elide this for brevity. Line 2 sets the preprocessor to ‘pca’ and Line 3 sets the classifier to a choice of ‘linear_svc’ or ‘random_forest’.

The syntaxes for the three AutoML tools in Figures 3, 4, and 5 differ. There are three ways to refer to the same operator: PCA, pca(-), and ‘pca’. There are three ways to specify an operator choice: a list of dictionaries, hp.choice, and a list of strings. The mental model varies from overwriting to nested configuration to string-based configuration. Users must learn new syntax and concepts for each tool and must rewrite code to switch tools. Moreover, as we consider more sophisticated pipelines (beyond the simple two-step one presented in the example), the search space specifications get even more convoluted and diverse between these existing specification schemes. A unified syntax would make these tools more consistent, easier to learn, and easier to switch. Furthermore, this syntax should unify not just AutoML tools but also be consistent with the manual end of the spectrum (P₁). More specifically, given that scikit-learn sets the de-facto standard for manual ML, the syntax should be scikit-learn compatible.

P₃: Support topology search and higher-order operators in AutoML tools. The tools previously discussed search operators and hyperparameters but do not optimize the topology of the pipeline itself. There are some tools that do, including TPOT [17], Recipe [8], and AlphaD3M [9]. Unfortunately, their methods for specifying the search space are inconsistent with manual machine learning and established tools. TPOT does not allow the user to specify the search space for pipeline topologies (the user can specify the set of operators and can fix the pipeline topology, disabling the topology search). Recipe and AlphaD3M use context-free grammars to specify the search space for the topology, but in a manner inconsistent with each other or with other levels of automation.

Some transformers (e.g., RFE in Figure 6 Line 7) and estimators (e.g., AdaBoostClassifier) are higher-order: they take other operators as arguments. Using the AutoML tools discussed so far to search inside their nested operators is not straightforward. The aforementioned TPOT, Recipe, and AlphaD3M do not handle higher-order operators in their search for pipeline topology.

A unified syntax for topology search and higher-order operators that is a natural extension of the syntax for manual machine learning, algorithm selection, and hyperparameter tuning would make AutoML more expressive while keeping it consistent.

P₄: Check for invalid configurations early and prune them out of search spaces. Even if the search for each hyperparameter uses a valid range in isolation, their combination can violate side constraints. Worse, these errors may be detected late, wasting time.

Figure 6 shows a misconfigured pipeline: the hyperparameters for LR in Line 7 are valid in isolation but invalid in combination. Unfortunately, this is not detected when the pipeline is created on Line 7. Instead, when Line 10 tries to fit the pipeline, it first fits the first step of the pipeline (see RFE in Line 6). Only then does it try to fit the LR and detect the mistake. This wastes 10 minutes (Line 15).

In contrast, a declarative specification of the side constraints would both speed up this manual example and enable AutoML search space generators to prune cases that violate constraints, thus speeding up the automated case too. Furthermore, in some situations, invalid configurations cause more harm than just wasted time, leading optimizers astray (Section 6). It is possible (with varying levels of difficulty) to incorporate these side constraints with the search space specification schemes used by the tools discussed earlier, but they each have inconsistent methods for doing this. Moreover, the complexity of these specifications makes them error prone. Additionally, while these side constraints help the optimizer, they do not directly help detect misconfigurations (as in Figure 6).

Custom validators would need to be written for each tool.

3 PROGRAMMING ABSTRACTIONS

This section shows LALE’s abstractions for consistent AutoML, addressing the problem statements P₁ ∧ P₂ ∧ P₃ ∧ P₄ from Section 2.

3.1 Abstractions for Declarative AutoML

An individual operator is a data science algorithm (aka. a primitive or a model), which may be a transformer or an estimator such as classifier or a regressor. Individual operators are modular building blocks from libraries such as scikit-learn. Figure 1 Lines 2–6 contain several examples, e.g., import LinearSVC as SVM. Mathematically, LALE views an individual operator as a function of the form

\[
\text{indivOp} : \theta_{\text{hyperparams}} \rightarrow D_{\text{fit}} \rightarrow D_{\text{in}} \rightarrow D_{\text{out}}
\]

This view uses currying: it views an operator as a sequence of functions each with a single argument and returning the next in the sequence. An individual operator (e.g., SVM) is a function from hyperparameters \(\theta_{\text{hyperparams}}\) (e.g., dual=False) to a function from training data \(D_{\text{fit}}\) (e.g., train_X, train_y) to a function from input data \(D_{\text{in}}\) (e.g., test_X) to output data \(D_{\text{out}}\) (e.g., predicted in Figure 1 Line 17). In the beginning, all arguments are latent, and each step in the sequence captures the next argument as given. The scikit-learn terminology for the three curried sub-functions is init, fit, and predict. Viewing operators as mathematical functions avoids complications arising from in-place mutation. It lets us conceptualize bindings as lifecycle: each bound, or captured, argument unlocks the functionality of the next state in the lifecycle.

A pipeline is a directed acyclic graph (DAG) of operators and a pipeline is itself also an operator. Since a pipeline contains operators and is an operator, it is highly composable. Furthermore,
viewing both individual operators and pipelines as special cases of operators makes the concepts more consistent. An example is make_pipeline(make_union(PCA, RFC), SVM), which is equivalent to (PCA & RFC) >> ConcatFeatures >> SWM. Here, & is the and combinator and >> is the pipe combinator. Combinators make edges more explicit and code more concise. An expression \( x \circ y \) composes \( x \) and \( y \) without introducing additional edges. An expression \( x \circ y \) introduces edges from all sinks of subgraph \( x \) to all sources of \( y \). Mathematically, Lale views a pipeline as a function of the form

\[
\text{pipeline} : \theta_{\text{topology}} \to \theta_{\text{hyperparams}} \to D_\text{fit} \to D_\text{in} \to D_\text{out}
\]

This uses currying just like individual operators, plus an additional \( \theta_{\text{topology}} \) at the start to capture the steps and edges. A pipeline is trainable if both \( \theta_{\text{topology}} \) and \( \theta_{\text{hyperparams}} \) are given, i.e., the hyperparameters of all steps have been captured. To fit a trainable pipeline, iterate over the steps in a topological order induced by the edges. For each step \( s \), let \( D^s_{\text{fit}} \) be the training data for the step, which is either the pipeline’s training data if \( s \) is a source or the predecessors’ output data \( D^s_{\text{fit}} = \{ D^p_{\text{fit}} \mid p \text{ pred}(s) \} \) otherwise. Then, recalling that \( s \) is a curried function, calculate \( s_{\text{trained}} = s(D^s_{\text{fit}}) \) and \( D^s_{\text{out}} = s_{\text{trained}}(D^s_{\text{fit}}) \). The trained pipeline substitutes trained steps for trainable steps in \( \theta_{\text{topology}} \). To make predictions with a trained pipeline, simply interpret \( \theta \) as dict to enable cloning.

An operator choice is an exclusive disjunction of operators and is itself also an operator. Operator choice specifies algorithm selection, and by being an operator, addresses problem \( P_1 \) from Section 2. An example is (SVM | RFC), where \( | \) is the or combinator. Mathematically, Lale views an operator choice as a function of the form

\[
opChoice : \theta_{\text{steps}} \to \theta_{\text{hyperparams}} \to D_\text{fit} \to D_\text{in} \to D_\text{out}
\]

This again uses currying. Argument \( \theta_{\text{steps}} \) is the list of operators to choose from. The \( \theta_{\text{hyperparams}} \) of an operator choice consists of an indicator for which of its steps is being chosen, along with the hyperparameters for that chosen step. Once \( \theta_{\text{hyperparams}} \) is captured, the operator choice is equivalent to just the chosen step, as shown in the visualization after Figure 1 Line 20.

The combined schema of an operator specifies the valid values along with search guidance for its latent arguments. It addresses problem \( P_3 \) from Section 2, supporting automated search with a pruned search space and early error checking all from the same single source of truth. Consider the pipeline \( \text{PCA} >> (J48 \mid LR) \), where PCA and LR are the principal component analysis and logistic regression from scikit-learn and J48 is a decision tree with pruning from Weka [11]. These operators have many hyperparameters and constraints and Lale handles all of them. For didactic purposes, this section discusses only a representative subset. Figure 7 shows the JSON Schema [18] specification of that subset. The open-source Lale library includes JSON schemas for many operators, some handwritten and others auto-generated [2]. The number of components for PCA is given by \( N \), which can be a continuous value in \((0.1)\) or the categorical value mle. The prior distribution helps AutoML tools search faster. J48 has a categorical hyperparameter \( R \) to enable reduced error pruning and a continuous confidence threshold \( C \) for pruning. Mathematically, we denote \( \text{allOf} \) as \( \land \), \( \text{anyOf} \) as \( \lor \), and \( \neg \) as \( \lnot \). Even though the valid values for \( C \) are \((0.1)\), search should only consider values \( C \in (0.05). \) The constraint in Lines 16–19 encodes a conditional \((R = \text{true}) \Rightarrow (C = 0.25)\) by using the equivalent \((R = \text{true}) \lor (C = 0.25). \) LR has two categorical hyperparameters \( S \) (solver) and \( P \) (penalty), with a constraint that solvers sag and lbfgs only support penalty l2, which we already encountered in Figure 6. Going forward, we will denote a JSON Schema object with properties as dict and a JSON Schema enum as [ ]. Eliding priors, this means Figure 7 becomes:

\[
\begin{align*}
\text{PCA} &: \text{dict}(N: (0.1) \lor \text{mle}) \\
J48 &: \text{dict}(R: \{\text{true}, \text{false}\}, C: (0.05)) \land \\
&\quad (\text{dict}(R: \text{true}) \Rightarrow \text{dict}(C: (0.25))) \\
LR &: \text{dict}(S: \{\text{linear}, \text{sag}, \text{lbfgs}\}, P: \{\text{l1}, \text{l2}\}) \land \\
&\quad (\text{dict}(S: \{\text{sag}, \text{lbfgs}\}) \Rightarrow \text{dict}(P: \{\text{l2}\}))
\end{align*}
\]

To auto-configure an operator means to automatically capture \( \theta_{\text{hyperparams}} \), which involves jointly selecting algorithms (for operator choices) and tuning hyperparameters (for individual operators). We saw examples for doing this with scikit-learn’s GridSearchCV [5], hyperopt-sklearn [15], and auto-sklearn [10] in Figures 3, 4, and 5. Lale offers a single unified syntax shown in Figure 1 Lines 15–16 to address problem \( P_2 \) from Section 2. Mathematically, let \( op \) be either an individual operator or a pipeline or choice whose \( \theta_{\text{topology}} \) or \( \theta_{\text{steps}} \) are already captured. That means \( op \) has the form \( \theta_{\text{hyperparams}} \to D_{\text{fit}} \to D_{\text{in}} \to D_{\text{out}}. \) Then auto_configure(op, \( D_{\text{fit}} \)) returns a function of the form \( D_{\text{in}} \to D_{\text{out}}. \) Section 4 discusses how to implement auto_configure using schemas.

### 3.2 Abstractions for Controlled AutoML

AutoML users rarely want to completely surrender all decisions to the automation. Instead, users typically want to control certain
decisions based on their problem statement and expertise. This section discusses how Lale supports such controlled AutoML.

The lifecycle state of an operator is the number of curried arguments it has already captured that determines the functionality it supports. To enable controlled AutoML, Lale pipelines can mix operators with different states. The visualizations in Figure 1 indicate lifecycle states via colors. A planned operator, shown in dark blue, has captured only \( \theta_{\text{topology}} \) or \( \theta_{\text{steps}} \), but \( \theta_{\text{hyperparams}} \) is still latent. Planned operators support \text{auto\_configure} but not \text{fit} or \text{predict}. A trainable operator, shown in light blue, has also captured \( \theta_{\text{hyperparams}} \), leaving \( D_{\text{Hi}} \) latent. Trainable operators support \text{auto\_configure} and \text{fit} but not \text{predict}. A trained operator, shown in white, also captures \( D_{\text{Hi}} \). Trained operators support \text{auto\_configure}, \text{fit}, and predict. Later states subsume the functionality of earlier states, enabling user control over where automation is applied. The state of a pipeline is the least upper bound of the states of its steps.

**Partially captured hyperparameters** of an individual operator treat some of its hyperparameters as given while keeping the remaining ones latent. Specifying some hyperparameters by hand lets users control and hand-prune the search space while still searching other hyperparameters automatically. For example, Figure 1 Line 12 shows partially captured hyperparameters for SVM and RFC. The visualization after Line 13 reflects this in a tooltip for RFC. And the pretty-printed code after Line 20 shows that automation respected the given dual hyperparameter of SVM while capturing the latent \( C \), penalty, and tol. Individual operators with partially captured hyperparameters are trainable: fit uses defaults for latents.

**Freezing** an operator turns future \text{auto\_configure} or \text{fit} operations into an identity. \( \text{PCA(n\_components=4).freeze\_trainable()} \gg \text{SVM} \) freezes all hyperparameters of PCA (using defaults for its latents), so \text{auto\_configure} on this pipeline tunes only the hyperparameters of SVM. Similarly, on a trained operator, freeze\_trained() freezes its learned coefficients, so any subsequent fit call will ignore its new \( D_{\text{Hi}} \). Freezing part of a pipeline speeds up (Auto-)ML.

A **custom schema** derives a variant of an individual operator that differs only in its schema. Custom schemas can modify ranges or distributions for search. As an extreme case, users can attach custom schemas to non-Lale operators to enable hyperparameter tuning on them—the call to \text{wrap\_imported\_operators} in Figure 1 Line 8 implicitly does that. Figure 8 shows how to explicitly customize a schema. Line 2 imports helper functions for expressing JSON Schema in Python. XGBoost implements a forest of boosted trees, so to get a small forest (a Grove), Line 5 restricts the number of trees to \([2..6]\). Line 6 restricts the booster to a constant, using a singleton enum. Afterwards, Grove is an individual operator like any other. Mathematically, we can view the ability to attach a schema to an individual operator as extending its curried function on the left:

\[
\text{indivOp} : \theta_{\text{schemas}} \to \theta_{\text{hyperparams}} \to D_{\text{Hi}} \to D_{\text{In}} \to D_{\text{Out}}
\]

### 3.3 Abstractions for Expressive AutoML

**A higher-order operator** is an operator that takes another operator as an argument. Scikit-learn includes several higher-order operators including RFE, AdaBoostClassifier, and BaggingClassifier. The nested operator is a hyperparameter of the higher-order operator and the nested operator can have hyperparameters of its own. Finding the best predictive performance requires AutoML to search both outside and inside higher-order operators. Figure 9 shows an example higher-order AdaBoostClassifier with a nested DecisionTreeClassifier. On the outside, AutoML should search the operator choice in Lines 2–3 and the hyperparameters of AdaBoostClassifier such as \( n_{\text{estimators}} \). On the inside, AutoML should tune the hyperparameters of DecisionTreeClassifier. Lale searches both jointly, helping solve problem \( P_3 \) from Section 2. The JSON schema of the base\_estimator hyperparameter of AdaBoostClassifier is:

```json
{ "description": "Base estimator from which the ensemble is built.",
  anyOf: [ [typeForOptimizer: "operator"], [enum: [null]]],
  default: null
}
```

A **pipeline grammar** is a context-free grammar that describes a possibly unbounded set of pipeline topologies. A grammar describes a search space for the \( \theta_{\text{topology}} \) and \( \theta_{\text{steps}} \) arguments needed to create planned pipelines and operator choices. Grammars formalize AutoML tools for topology search such as TPOT [17], Recipe [8], and AlphaD3M [9] that capture \( \theta_{\text{topology}} \) and \( \theta_{\text{steps}} \) automatically. Lale provides a grammar syntax that is a natural extension of concepts described earlier, helping solve problem \( P_3 \) from Section 2. Figure 10 shows a Lale grammar inspired by the AlphaD3M paper [9]. It describes linear pipelines comprising zero or more data cleaning operators, followed by zero or more transformers,
followed by exactly one estimator. This is implemented via recursive non-terminals: g.clean on Line 5 is a recursive definition, and so is g.tf.m on Line 6, implementing a search space with sub-pipelines of unbounded length. While Alpha3D3M uses reinforcement learning to search over this grammar, Figure 10 does something far less sophisticated. Line 13 unfolds the grammar to depth 3, obtaining a bounded planned pipeline, and Line 14 searches that using hyperopt, with no further modifications required.

Figure 11 shows a Lale grammar inspired by TPOT [17]. It describes possibly non-linear pipelines, which use not only \& but also the | combinator. Recall that (x \& y) | Concat applies both x and y to the same data and then concatenates the features from both. Besides transformers, Line 6 also uses estimators with | turning their predictions into features for downstream operators. This is not supported in scikit-learn pipelines but is supported in Lale.

Progressive disclosure is a design technique that makes things easier to use by only requiring users to learn new features when and as they need them. The starting point for Lale is manual machine learning, and thus, the scikit-learn code in Figure 2 is also valid Lale code. User needs to learn new features if they do not use AutoML. To use algorithm selection, users only need to learn about the | combinator and the auto_configure function. To express pipelines more concisely, users can learn about the \& and | combinators, but those are optional syntactic sugar for make_pipeline and make_union from scikit-learn. To use hyperparameter tuning, users only need to learn about wrap_imported_operators. To exercise more control over the search space, users can learn about freeze and well-documented standard proposal. To use higher-order operators, users need not learn new syntax, as Lale supports scikit-learn syntax for them. Finally, to use grammars, users need to add ‘g.‘ in front of their pipeline definitions; however, all the other features, such as the | combinator and the auto_configure function, continue to work the same with or without grammars.

### 4 SEARCH SPACE GENERATION

This section describes how to map the programming model from Section 3 to work directly with three popular AutoML tools: scikit-learn’s GridSearchCV [5], hyperopt [4], and SMAC [12], the library behind auto-sklearn [10].

---

**Figure 11: Example grammar inspired by TPOT.**

```plaintext
1: g = Grammar()
2: g.start = g.process >> g.features >> g.model
3: g.process = (g.feature1 | (g.feature1 & g.process) >> Concat)
4: g.feature1 = g.feature1 | g.est & g.features >> Concat
5: g.model = g.est

Figure 11: Example grammar inspired by TPOT.
For working with higher-order operators, Lale adds additional markers that enable it to reconstruct nested operators later.

The SMAC backend has to flatten Lale’s nested IR into a grid of disjuncts with discriminants D. To do this, it internally uses a name mangling encoding that extends the ___mangling of scikit-learn, an example for which occurs in Figure 3 Line 7. Each element of the grid needs to be a simple ‘dict’ with no further nesting. For our running example, the result in mathematical notation is:

\[
\begin{align*}
\text{dict}(N: (0..1), D: [J48], R: [false], C: (0..0.5)) \\
\lor \text{dict}(N: (0..1), D: [J48], R: [true, false], C: [0.25]) \\
\lor \text{dict}(N: [mle], D: [J48], R: [false], C: (0..0.5)) \\
\lor \text{dict}(N: [mle], D: [J48], R: [true, false], C: [0.25]) \\
\lor \text{dict}(N: (0..1), D: [LR], S: [linear], P: [\{I1, I2\}]) \\
\lor \text{dict}(N: (0..1), D: [LR], S: [linear, sag, lbfgs], P: [\{I2\}]) \\
\lor \text{dict}(N: [mle], D: [LR], S: [linear], P: [\{I1, I2\}]) \\
\lor \text{dict}(N: [mle], D: [LR], S: [linear, sag, lbfgs], P: [\{I2\}])
\end{align*}
\]

Next, the SMAC backend adds conditionals that tell the Bayesian optimizer which variables are relevant for which disjunct, and prior distribution adds Python 3 type hints so users can get additional help from tools such as MyPy, PyCharm, or VSCode.

To implement the combiners >>, & and 1, Lale uses Python’s overloaded __rshift__, __and__, and __or__ methods. Python only supports overriding these as instance methods. Therefore, unlike in scikit-learn, Lale planned operators are object instances, not classes. This required emulating the scikit-learn __init__ with __call__.

The implementation carefully avoids in-place mutation of operators by methods such as auto_configure, fit, customize_schema, or unfold. This prevents unintended side effects and keeps the implementation consistent with the mathematical function abstractions from Section 3.1. Unfortunately, in scikit-learn, fit does in-place mutation, so for compatibility, Lale supports that but with a deprecation warning.

6 RESULTS

This section evaluates Lale on OpenML classification tasks and on different data modalities. It also experimentally demonstrates the importance of side constraints for the optimization process. For each experiment, we specified a Lale search space and then used auto_configure to run hyperopt on it. The value proposition of Lale is to leverage existing AutoML tools effectively and consistently; in general, we do not expect Lale to outperform them.

6.1 Benchmarks: OpenML Classification

To demonstrate the use of Lale, we designed four experiments that specify different search spaces for OpenML classification tasks.

lale-pipe: The three-step planned lale_openml_pipeline in Figure 12.

lale-ad3m: The AlphaD3M-inspired grammar of Figure 10 unfolded with a maximal depth of 3.

lale-tpot: The TPOT-inspired grammar of Figure 11 unfolded with a maximal depth of 3.

lale-ad: The higher-order operator pipeline of Figure 9.

For comparison, we used auto-sklearn [10] — a popular scikit-learn based AutoML tool that has won two OpenML challenges — with its default setting as a baseline. We chose 15 OpenML datasets for which we could get meaningful results (more than 30 trials).
using the default settings of Auto-sklearn. The selected datasets comprise 5 simple classification tasks (test accuracy > 90% in all our experiments) and 10 harder tasks (test accuracy < 90%). 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, and for each task, the total optimization time was set to 1 hour with a timeout of 6 minutes per trial.

Table 1 presents the results of our experiments. For each experiment, we report the test accuracy of the best pipeline found averaged over 5 runs. Note that for the shuttle dataset, 3 out of 5 runs of auto-sklearn resulted in a MyDummyClassifier being returned as the result. Since we were trying to evaluate the default settings, we did not attempt debugging it, but according to the tool’s issue log, other users have encountered it before. The column askl-adb reports the accuracy of auto-sklearn when the set of classifiers is limited to AdaBoost with only data preprocessing. These results are presented for comparison with the lale-adb experiments, as the data preprocessing operators in Figure 9 were chosen to match those of auto-sklearn as much as possible. Also note that the default setting for auto-sklearn uses meta-learning.

The results show that carefully crafted search spaces (e.g., TPOT-inspired grammar, or pipelines with higher-order operators) and off-the-shelf optimizers such as hyperopt can achieve accuracies that are competitive with state-of-the-art tools. This raises the question of how important those side constraints are in practice.

### 6.2 Case Studies: Other Modalities

While all the examples so far focused on tasks for tabular datasets, the core contribution of Lale is not limited to those. This section demonstrates Lale’s versatility on three datasets from different modalities. Table 2 summarizes the results.

**Text.** We used the Drug Review dataset for predicting a rating given by a patient to a drug. The Drug Review dataset has a text column called review, to which the pipeline applies either TfidfVectorizer from scikit-learn or a pretrained BERT embeddings, which is a text embedding based on neural networks. The dataset also has numeric columns, which are concatenated with the result of the embedding.

**Image.** We used the CIFAR-10 computer vision dataset. We picked the ResNet50 deep-learning model, since it has been shown to do well on CIFAR-10. Our experiments kept the architecture of ResNet50 fixed and tuned learning-procedure hyperparameters.

**Time-series.** We used the Epilepsy dataset, a subset of the TUH Seizure Corpus, for classifying seizures by onset location (generalized or focal). We used a three-step pipeline:

We implemented a popular pre-processing method [19] in a Window operator with three hyperparameters $W$, $O$, and $T$. Note that this transformer leads to multiple samples per seizure. Hence, during evaluation, each seizure is classified by taking a vote of the predictions made by each sample generated from it.

### 6.3 Effect of Side Constraints on Convergence

Lale’s search space compiler takes rich hyperparameter schemas including side constraints and translates them into semantically equivalent search spaces for different AutoML tools. This raises the question of how important those side constraints are in practice.
To explore this, we did an ablation study where we generated not just the constrained search spaces that are default with Lale but also unconstrained search spaces that drop side constraints. With hyperopt on the unconstrained search space, some iterations are unsuccessful due to exceptions, for which we reported np.float.max loss. Figure 13 plots the convergence for the Car dataset on the planned pipeline LR | KNN. Both of these operators have a few side constraints. Whereas the unconstrained search space causes some invalid points early in the search, the two curves more-or-less coincide after about two dozen iterations. The story looks very different in Figure 14 when adding a third operator J48 | LR | KNN. In the unconstrained case, J48 has many more invalid runs, causing hyperopt to see so many np.float.max loss values from J48 that it gives up on it. In the constrained case, on the other hand, J48 has no invalid runs, and hyperopt eventually realizes that it can configure J48 to obtain substantially better performance.

6.4 Dataset Details
In order to be specific about the exact datasets for reproducibility, Table 3 report the URLs for accessing those.

7 RELATED WORK
There are various search tools, designed around scikit-learn [5], and each usually focused on a particular novel optimization algorithm. Auto-sklearn [10] uses the search space specification and optimization algorithm of SMAC [12]. Hyperopt-sklearn [15] uses its own search space specification scheme and a novel optimizer based on Tree-structured Parzen Estimators (TPE) [3]. Scikit-learn also comes with its own GridSearchCV and RandomizedSearchCV classes. Auto-Weka [20] is a predecessor of auto-sklearn that also uses SMAC but operates on operators from Weka [11] instead of scikit-learn. TPOT [17] is designed around scikit-learn and uses genetic programming to search for pipeline topologies and operator choices. This usually leads to the generation of many misconfigured pipelines, wasting execution time. RECIPE [8] prunes away the misconfigurations to save execution time by validating generated pipelines with a grammar for linear pipelines. AlphaD3M [9] makes use of a grammar in a generative manner (instead of just for validation) with a deep reinforcement learning based algorithm. Katz et al. [13] similarly use a grammar for Lale pipelines with AI planning tools. In contrast to these tools, the contribution of this paper is not a novel optimization algorithm but rather a more consistent programming model with search space generation targeting existing tools.

8 CONCLUSION
This paper describes Lale, a library for semi-automated data science. Lale contributes a syntax that is consistent with scikit-learn, but extends it to support a broad spectrum of automation including algorithm selection, hyperparameter tuning, and topology search. Lale works by automatically generating search spaces for established AutoML tools, extending their capabilities to grammar-based search and to search inside higher-order operators. The experiments show that search spaces crafted using Lale achieve results that are competitive with state-of-the-art tools while offering more versatility.
REFERENCES

[1] Guillaume Baudart, Martin Hirzel, Kiran Kate, Louis Mandel, and Avraham Shinnar. 2019. Machine Learning in Python with No Strings Attached. In Workshop on Machine Learning and Programming Languages (MAPL). 1–9. https://doi.org/10.1145/3315508.3329972

[2] Guillaume Baudart, Peter Kirchner, Martin Hirzel, and Kiran Kate. 2020. Mining Documentation to Extract Hyperparameter Schemas. In ICML Workshop on Automated Machine Learning (AutoML@ICML). https://arxiv.org/abs/2006.16984

[3] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balazs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In Conference on Neural Information Processing Systems (NIPS). http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization

[4] James Bergstra, Brent Komer, Chris Eliaismith, Dan Yamins, and David D. Cox. 2015. Hyperopt: a Python Library for Model Selection and Hyperparameter Optimization. Computational Science & Discovery 8, 1 (2015). http://dx.doi.org/10.1088/1749-4699/8/1/014008

[5] Lars Butinck, 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. 2013. API Design for Machine Learning Software: Experiences from the scikit-learn Project. https://arxiv.org/abs/1309.0238

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In Conference on Knowledge Discovery and Data Mining (KDD). 785–794. http://doi.acm.org/10.1145/2939672.2939785

[7] Noam Chomsky. 1956. Three models for the description of language. IRE Transactions on Information Theory 2, 3 (1956), 113–124.

[8] Alex G. C. de Sá, Walter José G. S. Pinto, Luiz Otavio V. B. Oliveira, and Gisele L. Pappa. 2015. Hyperopt-Sklearn: Automated Hyperparameter Configuration for Scikit-Learn. In Python in Science Conference (SciPy). 32–37. http://conference.scipy.org/proceedings/scipy2014/komer.html

[9] Iddo Drori, Yamuna Krishnamurthy, Raoni Lourenco, Remi Rampin, Kyunghyun Cho, Claudio Silva, and Juliana Freire. 2019. Automatic Machine Learning by Foundations of JSON Schema. In European Conference on Genetic Programming (EuroGP). 263–273. https://doi.org/10.1007/978-3-95981-059-9_28

[10] Kaspar Schindler, Howan Leung, Christian E. Elger, and Klaus Lehnertz. 2006. Assessing seizure dynamics by analysing the correlation structure of multichannel intracranial EEG. Brain 130, 1 (11 2006), 65–77. arXiv:http://oup.prod.sis.lan/brain/article-pdf/130/1/65/992272/awl304.pdf https://doi.org/10.1093/brain/awl304

[11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithm for Hyper-Parameter Optimization. In Conference on Neural Information Processing Systems (NIPS). 3146–3154. http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

[12] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. 2020. Exploring Cross-Field Languages via Planning: The Case for Automating Machine Learning. In International Conference on Automated Planning and Scheduling (ICAPS), Vol. 30. 403–411. https://www.aaai.org/ojs/index.php/ICAPS/article/view/6868

[13] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In International Conference on Learning and Intelligent Optimization (LION). 507–523. https://doi.org/10.1007/978-3-642-25566-3_40

[14] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. 2020. Exploring Cross-Field Languages via Planning: The Case for Automating Machine Learning. In International Conference on Automated Planning and Scheduling (ICAPS), Vol. 30. 403–411. https://www.aaai.org/ojs/index.php/ICAPS/article/view/6868

[15] Brent Komer, James Bergstra, and Chris Eliaismith. 2014. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. In Python in Science Conference (SciPy). 32–37. http://conference.scipy.org/proceedings/scipy2014/komer.html

[16] Alex G. C. de Sá, Walter José G. S. Pinto, Luiz Otavio V. B. Oliveira, and Gisele L. Pappa. 2015. Hyperopt-Sklearn: Automated Hyperparameter Configuration for Scikit-Learn. In Python in Science Conference (SciPy). 32–37. http://conference.scipy.org/proceedings/scipy2014/komer.html

[17] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. 2020. Exploring Cross-Field Languages via Planning: The Case for Automating Machine Learning. In International Conference on Automated Planning and Scheduling (ICAPS), Vol. 30. 403–411. https://www.aaai.org/ojs/index.php/ICAPS/article/view/6868

[18] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. 2020. Exploring Cross-Field Languages via Planning: The Case for Automating Machine Learning. In International Conference on Automated Planning and Scheduling (ICAPS), Vol. 30. 403–411. https://www.aaai.org/ojs/index.php/ICAPS/article/view/6868

[19] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Laven-