Efficient End-to-End AutoML via Scalable Search Space Decomposition (Extended Paper)

Yang Li*, Yu Shen*, Wentao Zhang*, Ce Zhang[†], Bin Cui

Abstract End-to-end AutoML has attracted intensive interests from both academia and industry which automatically searches for ML pipelines in a space induced by feature engineering, algorithm/model selection, and hyper-parameter tuning. Existing AutoML systems, however, suffer from scalability issues when applying to application domains with large, high-dimensional search spaces. We present VOLCANOML, a scalable and extensible framework that facilitates systematic exploration of large AutoML search spaces. VOLCANOML introduces and implements basic building blocks that decompose a large search space into smaller ones, and allows users to utilize these building blocks to compose an *execution plan* for the AutoML problem at hand. VOLCANOML further supports a Volcano-style *execution model* – akin to the one supported by modern database systems – to execute the plan constructed. Our evaluation demonstrates that, not only does VOLCANOML raise the level of expressiveness for search space decomposition in AutoML, it also leads to actual findings of decomposition strategies that are significantly more efficient than the ones employed by state-of-the-art AutoML systems such as `auto-sklearn`. This paper is the extended version of the initial VolcanoML paper appeared in VLDB 2021.

*Yang Li
Institute: School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
Institute: Tencent data platform, TEG, Tencent inc.
E-mail: liyang.cs@pku.edu.cn

*Yu Shen
Institute: School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
E-mail: shenyu@pku.edu.cn

*Wentao Zhang
Institute: School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
Institute: Tencent data platform, TEG, Tencent inc.
E-mail: wentao.zhang@pku.edu.cn

[†] Ce Zhang
Institute: Department of Computer Science, ETH Zürich
E-mail: ce.zhang@inf.ethz.ch

Corresponding author: Bin Cui
Institute: Department of Computer Science and Technology & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
Tel: +86-10-62765821
E-mail: bin.cui@pku.edu.cn

1 Introduction

In recent years, researchers in the database community have been working on raising the level of abstractions of machine learning (ML) and integrating such functionality into today's data management systems [95, 96], e.g., SystemML [25], SystemDS [8], Snorkel [71], ZeroER [91], TFX [5,9], Query 2.0 [92], Krypton [66], Cerebro [67], ModelDB [86], MLFlow [94], Deep-Dive [14], HoloClean [72], EaseML [1], ActiveClean [48], and NorthStar [47]. End-to-end AutoML systems [93, 97,33] have been an emerging type of systems that has significantly raised the level of abstractions of building ML applications. Given an input dataset and a user-defined utility metric (e.g., validation accuracy), these systems automate the search of an end-to-end ML pipeline, including *feature engineering*, *algorithm/model selection*, and *hyper-parameter tuning*. Open-source examples include `auto-sklearn` [22], `TPOT` [69], and `hyperopt-sklearn` [46], whereas most cloud service providers, e.g., Google, Microsoft, Amazon, etc., all

provide their proprietary services on the cloud. As machine learning has become an increasingly indispensable functionality integrated in modern data (management) systems, an efficient and effective end-to-end AutoML component also becomes increasingly important.

End-to-end AutoML provides a powerful abstraction to automatically navigate and search in a given complex search space. However, in our experience of applying state-of-the-art end-to-end AutoML systems in a range of real-world applications [2], we find that such a system running fully automatically is rarely enough — often, developing a successful ML application involves multiple iterations between a user and an AutoML system to iteratively improve the resulting ML artifact.

Motivating Practical Challenge. One such type of interaction, which inspires this work, is the *enrichment of search space*. We observe that the default search space provided by state-of-the-art AutoML systems is often not enough in many applications. This was not obvious to us at all in the beginning and it is not until we finish building a range of real-world applications that we realize this via a set of concrete examples. For example, in one of our astronomy applications [75], the feature normalization function is domain-specific and not supported by most, if not all, AutoML systems. Similar examples can also be found when searching for suitable ML models via AutoML. In one of our meteorology applications, we need to extend the models with meteorology-specific loss functions. We saw similar problems when we tried to extend existing AutoML systems with pre-trained feature embeddings coming from TensorFlow Hub, to include include models that have been newly published on arXiv to enrich the Model Base [52], or to support Cosine annealing as for tuning.

Technical Challenge: Scalability over the Search Space. "*Why is it hard to extend the search space, as a user, in an end-to-end AutoML system?*" The answer to this question is a complex one that is *not* completely technical: some aspects are less technical such as engineering decisions and UX designs, however, there are also more *technically fundamental* aspects. An end-to-end AutoML system contains an optimization algorithm that navigates a joint search space induced by *feature engineering*, *algorithm selection*, and *hyper-parameter tuning*. Because of this joint nature, the search space of end-to-end AutoML is complex and huge while the enrichment is only going to make it even larger. As we will see, handling such a huge space is already challenging for existing systems, and further enriching it will make it even harder to scale.
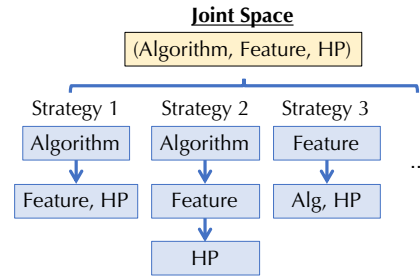

Fig. 1 Different decomposition choices.

Many existing systems such as `auto-sklearn` [22] and `TPOT` [69] deal with the entire composite search space *jointly*, which naturally leads to the scalability bottleneck. Decomposing a joint space has been explored for some subspaces (e.g., only algorithm and hyper-parameters as in [63, 53]), however, none of them has been applied to a search space as large as that of end-to-end AutoML. One challenge is that there exist many different ways to decompose the same space (See Figure 1), as shown above, but only *some* of them can perform well. *Without a structured, high-level abstraction for search space decomposition to explore different strategies, it is very hard to scale up an end-to-end AutoML system to accommodate the search space that will only get larger in the future.*

Summary of Contributions. The initial version of this paper [59] appeared in VLDB 2021, where we focused on designing the system, VOLCANOML, which is *scalable to a large search space*. In this paper, we make the following four additional contributions: First, we provide the automatic execution plan generation module (in Section 4.2) to enrich the proposed framework, and discuss the advantages and underlying problems in this direction. Second, we propose the meta-learning based components for the building blocks (in Section 5) to further speed up VOLCANOML. Third, we conduct a comprehensive set of experiments (in Section 6) to demonstrate the effectiveness and efficiency of VOLCANOML, and provide the results about automatic plan generation and meta-learning based acceleration. Finally, we provide more details about system components, implementations (interfaces) and search spaces in Section A of the appendix. Our technical contributions are as follows.

C1. System Design: A Structured View on Decomposition. The main technical contribution of VOLCANOML is to provide a flexible and principled way of decomposing a large search space into multiple smaller ones. We propose a novel system abstraction: a set of VOLCANOML *building blocks* (Section 3), each of which

takes charge of a smaller sub-search space whereas a VOLCANOML *execution plan* (Section 4) consists of a *tree* of such building blocks — the root node corresponds to the original search space and its child nodes correspond to different subspaces. Under this abstraction, optimizing in the joint space is conducted as optimization problems over different smaller subspaces. The execution model is similar to the classic Volcano query evaluation model in a relational database [24] (thus the name VOLCANOML): the system asks the root node to take one iteration in the optimization process, which *recursively* invokes one of its child nodes to take one iteration on solving a smaller-scale optimization problem over its own subspace; this recursive invocation procedure will continue until a leaf node is reached. This flexible abstraction allows us to explore different ways that the same joint space can be decomposed. Together with the meta-learning based optimizations (Section 5), VOLCANOML can often support more scalable search process than the existing AutoML systems such as `auto-sklearn` and `TPOT`.

C2. Large-scale Empirical Evaluations. We conducted intensive empirical evaluations, comparing VOLCANOML with state-of-the-art systems including `auto-sklearn` and `TPOT`. We show that (1) under the *same search space* as `auto-sklearn`, VOLCANOML significantly outperforms `auto-sklearn` and `TPOT` — over 30 classification tasks and 20 regression tasks — VOLCANOML outperforms the *best* of `auto-sklearn` and `TPOT` on a majority of tasks; concretely, VOLCANOML could achieve a higher balanced accuracy for classification tasks and a smaller mean square error for regression tasks given the same time budget; (2) using an *enriched search space* with additional feature engineering operators, VOLCANOML performs significantly better than `auto-sklearn`; (3) using an *enriched search space* with an additional data processing stage and functionalities beyond what `auto-sklearn` and `TPOT` currently support (i.e., an additional embedding selection stage using pre-trained models on TensorFlow Hub), VOLCANOML can deal with input types such as images efficiently; and (4) VOLCANOML is at least comparable with and often outperforms four industrial AutoML platforms on six Kaggle competitions.

Moving Forward. The VOLCANOML abstraction enables a structured view of optimizing a black-box function via decomposition. This structured view itself opens up interesting future directions. For example, one may wish to *automatically* decompose a search space

given a workload, just like what a classic query optimizer would do for relational queries. For constrained optimizations, we also imagine techniques similar to traditional "*push-down selection*" could be applied in a similar spirit. We explore the possibility of automatically searching for the best plan in Section 4 and discuss the limitations of this simple strategy and the exciting line of future work that could follow. While the full treatment of these aspects are beyond the scope of this paper, we hope the VOLCANOML abstraction can serve as a foundation for these future endeavors.

2 Related Work

AutoML is a topic that has been intensively studied over the last decade. We briefly summarize related work in this section and readers can consult latest surveys [33,93,97,29] for more details.

End-to-End AutoML. End-to-end AutoML, the focus of this work, aims to automate the development process of the end-to-end ML pipeline, including feature preprocessing, feature engineering, algorithm selection, and hyper-parameter tuning. Often, this is modeled as a black-box optimization problem [34] and solved jointly [22,82,69]. Apart from grid search and random search [6], genetic programming [64,69] and Bayesian optimization (BO) [7,32,79,20,77] has become prevailing frameworks for this problem. One challenge of end-to-end AutoML is the staggeringly huge search space that one has to support and many of these methods suffer from scalability issues [57,55]. In addition, meta-learning [84,56,23] systematically investigates the interactions that different ML approaches perform on a wide range of learning tasks, and then learns from this experience, to accomplish new tasks much faster. Several meta-learning approaches [74,31,83,22,54] can guide ML practitioners to design better search spaces for AutoML tasks.

Many end-to-end AutoML systems have raised the abstraction level of ML. `auto-weka` [82], `hyperopt-sklearn` [46], and `auto-sklearn` [22] are the main representatives of BO-based AutoML systems. `auto-sklearn` is one of the most popular open-source frameworks. `TPOT` [69] and ML-Plan [64] use genetic algorithms and hierarchical task networks planning, respectively, to optimize over the pipeline space, and require discretization of the hyper-parameter space. AlphaD3M [18] integrates reinforcement learning with Monte Carlo tree search (MCTS) to solve AutoML problems but without imposing efficient decomposition over hyper-parameters and algorithm selection. AutoStacker [13] focuses on ensembling and cascading

to generate complex pipelines, and solves the CASH (Combined Algorithm Selection and Hyperparameters optimization) problem [22] via random search. ML Bazaar [78] is a general-purpose, multi-task, end-to-end AutoML system, which pair ML pipelines with a hierarchy of AutoML strategies – Bayesian optimization. Furthermore, a growing number of commercial enterprises also export their AutoML services to their users, e.g., `H2O` [49], Microsoft's Azure Machine Learning [4], Google Cloud's AI Platform [27], Amazon Machine Learning [61] and IBM's Watson Studio AutoAI [35].

Automating Individual Components. Apart from end-to-end AutoML, many efforts have been devoted to studying sub-problems in AutoML: (1) feature engineering [44, 42, 41, 68, 43], (2) algorithm selection [82, 46, 22, 19, 63, 53], and (3) hyper-parameter tuning [32, 79, 7, 51, 36, 21, 57, 80, 45, 39, 70, 30, 76, 90, 37]. Meta-learning methods [89, 26, 23] for hyper-parameter tuning can leverage auxiliary knowledge acquired from previous tasks to achieve faster optimization. Several systems offer a subset of functionalities in the end-to-end process. Microsoft's NNI [73] helps users to automate feature engineering, hyper-parameter tuning, and model compression. Recent work [63] leverages the ADMM optimization framework to decompose the CASH problem [22], and solves two easier sub-problems. Berkeley's Ray [65] and OpenBox [58] provide the `tune` module [60, 57] to support scalable hyper-parameter tuning tasks in a distributed environment. Featuretools [40] is a Python library for automatic feature engineering. Unlike these works, in this paper, we focus on deriving an end-to-end solution to the AutoML problem, where the sub-problems are solved in a joint manner.

Volcano Model. The Volcano model [28] (originally known as the Iterator Model) is the classical evaluation strategy of an analytical DBMS query: Each relational-algebraic operator produces a tuple stream, and a consumer can iterate over its input streams. The tuple stream has three interfaces: `open`, `next` and `close`; all operators own the same interface, and the implementation is opaque. It is a chain of iterators and data flows through them when the topmost iterator calls `next()` on the iterator below it. This results in propagation of `next()` calls till the bottom-most iterator is called.

3 VolcanoML and Building Blocks

The goal of VOLCANOML is to enable scalability with respect to the underlying AutoML search space. As a result, its design focuses on the *decomposition* of a
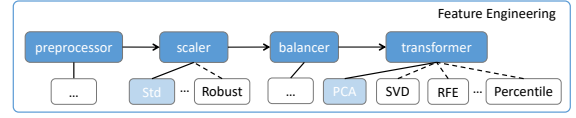

Fig. 2 The search space of FE pipeline.

given search space. In this section, we first introduce key building blocks in VOLCANOML, and in Section 4 we describe how multiple building blocks are put together to compose a VOLCANOML *execution plan* in a modular way. Later in Section 5, we introduce additional optimizations for these building blocks.

3.1 Search Space of End-to-End AutoML

We describe the search space of end-to-end AutoML following the presentation in `auto-sklearn`[22]. The input to the system is a dataset D, containing a set of training samples. The user also provides a pre-defined metric, e.g., validation accuracy or cross-validation accuracy, to measure the *utility* of a given ML pipeline. The output of an end-to-end AutoML system is an ML pipeline that achieves good utility.

To find such an ML pipeline, the system searches over a large search space of possible pipelines and picks one that maximizes the pre-defined utility. This search space is a composition of (1) feature engineering operators, (2) ML algorithms/models, and (3) hyper-parameters.

Feature Engineering. The feature engineering process takes as input a dataset D and outputs a new dataset D'. It achieves this by transforming the input dataset via a set of data transformations. The pipeline for feature engineering is shown in Figure 2. It comprises four sequential stages: *preprocessors* (compulsory), *scalers* (5 possible operators), *balancers* (1 possible operators) and *feature transformers* (13 possible operstors). For each stage, the system chooses a single transformation to apply. For example, for *feature_transforming*, the system can choose among `no_processing`, `kernel_pca`, `polynomial`, `select_percentile`, etc.

ML Algorithms. Given a transformed dataset D', the system then picks an ML algorithm to train. Since different ML algorithms are suitable for different types of tasks, the system needs to consider a diverse range of possible ML algorithms. Taking `auto-sklearn` as an example, the search space for ML algorithms

contains `Linear_Model`, `Support_Vector_Machine`, `Discriminant_Analysis`, `Random_Forest`, etc.

Hyper-parameters. Each ML algorithm has its own sub-search space for hyper-parameter tuning — if we choose to use a certain ML algorithm, we also have to specify the corresponding hyper-parameters. The hyper-parameters fall into three categories: continuous (e.g., `sub-sample_rate` for `Random_Forest`), discrete (e.g., `maximal_depth` for `Decision_Tree`), and categorical (e.g., `kernel_type` for `Lib_SVM`).

(AutoML optimization.) If the system makes a concrete pick for each of the above decisions, then it can compose a concrete ML pipeline and evaluate its utility. Concretely, given a pipeline configuration that determines the details of feature engineering, algorithm and hyperparameters, we could construct a specific ML pipeline. Then we need to train a corresponding ML model within this pipeline, and evaluate its performance on the validation set to obtain the utility of this pipeline configuration. This is often an expensive process since it involves training an ML model. To find the optimal ML pipeline, the system evaluates the utility of different ML pipelines in an iterative manner following a *search strategy*, and picks the one that maximizes the utility. The candidates for search strategy can be random search [6], grid search, genetic algorithms [64,69], Bayesian optimization [7,32,79], bandits based methods [36,51], etc.

For example, `auto-sklearn` handles the above search space *jointly* and optimizes it with Bayesian optimization (BO) [77]. Given an initial set of function evaluations, BO proceeds by fitting a surrogate model to those observations, specifically a *probabilistic Random Forest* in `auto-sklearn`, and then chooses which ML pipeline to evaluate from the search space by optimizing an acquisition function that balances exploration and exploitation.

3.2 Building Blocks

Unlike `auto-sklearn`, VOLCANOML decomposes the above search space into smaller subspaces. **(Key idea.)** Instead of searching over a huge pipeline space, it could be easier for an algorithm to optimize over its subspaces. Decomposing a joint space has been explored in many domains [63,53]. The way how to decompose the pipeline space into subspaces in field of AutoML is still remains open. Next, we propose a structured and high-level abstraction to support scalable search space decomposition. One interesting design decision in VOLCANOML is to introduce a *structured abstraction* to express different *decomposition strategies*. A de-

composition strategy is akin to an *execution plan* in relational database management systems, which is composed of *building blocks* akin to relational operators. A building block itself can be viewed as an *atomic* decomposition strategy. We next present the details of the building blocks implemented by VOLCANOML, and we will introduce how to use these blocks to compose VOLCANOML execution plans in Section 4.

Goal. The *goal* of VOLCANOML is to solve:

$$\min_{x_1,...,x_n} f(x_1, ..., x_n; D), \tag{1}$$

where $x_1, ..., x_n$ is a set of n variables and each of them has domain $\mathbb{D}_{x_i}$ for $i \in [n]$. Together, these n variables define a search space $(x_1, ..., x_n) \in \prod_i \mathbb{D}_{x_i}$. D corresponds to the input dataset, which is a *set* of input samples. In AutoML, the variables $x_1, ..., x_n$ are actually the pipeline hyperparameters, and the search space is the complete pipeline search space, which is a composition of feature engineering operators, ML algorithms/models, and hyper- parameters. The optimization objective f is to minimize the validation loss (e.g., classification error), i.e., the objective function $f(\cdot)$ in Formula 1. In our setting, $f(\cdot)$ is a black-box function that we can only evaluate (but not exploiting the derivative), and the objective is to solve $\min f(\cdot)$ as quickly as possible. Given a fixed $\boldsymbol{c}$ (i.e., a concrete ML pipeline) in the composite domain $\boldsymbol{c} \in \prod_i \mathbb{D}_{x_i}$, we use the notation $f(\boldsymbol{c}; D)$ as the value of evaluating f by substituting $(x_1, ...x_n)$ with $\boldsymbol{c}$.

Subgoal. One key decision of VOLCANOML is to solve the optimization problem on a search space by decomposing it into multiple smaller subspaces, each of which will be solved by one *building block*. We define optimizing over each of these smaller subspaces as a *subgoal* of the original problem. Formally, a subgoal g is defined by two components: $\bar{\boldsymbol{x}}_g \subseteq \{x_1, ...x_n\}$ as a subset of variables, and $\bar{\boldsymbol{c}}_g \in \prod_{x_i \in \bar{\boldsymbol{x}}_g} \mathbb{D}_{x_i}$ as an assignment in the domain of all variables in $\bar{\boldsymbol{x}}_g$. Let $\bar{\boldsymbol{x}}_{-g} = \{x_1, ..., x_n\} - \bar{\boldsymbol{x}}_g$ be all variables that are *not* in $\bar{\boldsymbol{x}}_g$.

Each subgoal defines a function f_g over a smaller search space, which is constructed by *substituting* all variables in $\bar{\boldsymbol{x}}_g$ with $\bar{\boldsymbol{c}}_g$:

$$\begin{aligned} f_g =& f[\bar{\boldsymbol{x}}_g / \bar{\boldsymbol{c}}_g] : \\ & \boldsymbol{z} \in \prod_{x_i \in \bar{\boldsymbol{x}}_{-g}} \mathbb{D}_{x_i} \mapsto f(\{\bar{\boldsymbol{c}}_g; \boldsymbol{z}\}; D). \end{aligned} \tag{2}$$

Each subgoal is a sub-problem in the ML pipeline search of AutoML such as feature engineering, algorithm selection, etc.

Building Block. Each subgoal g corresponds to one building block $B_{g,D}$, whose goal is to solve

$$\min_{\bar{\boldsymbol{x}}_{-g}} f_g(\bar{\boldsymbol{x}}_{-g}; D). \tag{3}$$

A building block $B_{g,D}$ imposes several assumptions on g and D. First, given an assignment $\bar{\boldsymbol{c}}_{-g}$ to $\bar{\boldsymbol{x}}_{-g}$, it is able to evaluate the value of the function $f_g(\bar{\boldsymbol{c}}_{-g}, D)$. Second, given a dataset D, a building block has the knowledge about how to subsample a smaller dataset $\tilde{D} \subseteq D$ and then conduct evaluations on such a subset $\boldsymbol{x} \mapsto f_g(\boldsymbol{x}; \tilde{D})$. Third, we assume that the building block has access to a cost model about the cost of an evaluation at $\boldsymbol{x}$, $C_{g,D,\boldsymbol{x}}$.

Interfaces. All implementations of a building block follow an interactive optimization process. A building block exposes several interfaces. First, one can initialize a building block via

$$B_{g,D} \leftarrow \texttt{init}(f, \bar{\boldsymbol{x}}_g, \bar{\boldsymbol{c}}_g, D), \tag{4}$$

which creates a building block (i.e., a new subproblem). Second, one can query the current best solution found in $B_{g,D}$ by

$$\hat{\boldsymbol{x}} \leftarrow \texttt{get\_current\_best}(B_{g,D}). \tag{5}$$

Furthermore, one can ask $B_{g,D}$ to iterate once via

$$\texttt{do\_next!}(B_{g,D}), \tag{6}$$

where '!' indicates potential change on the state of the input $B_{g,D}$.

Last but not least, one can query a building block about its _expected utility_ (EU) if given K more budget units (e.g., seconds) via

$$[l, u] \leftarrow \texttt{get\_eu}(B_{g,D}, K). \tag{7}$$

By adopting a similar design principle used in the existing AutoML systems [22,69,63], in VolcanoML we estimate EU by _extrapolation_ into the future with more available budget. Given the inherent uncertainty in our estimation method, rather than returning a single point estimate, we instead return a lower bound l and an upper bound u. We refer readers to [53] for the details of how the lower and upper bounds are established. Moreover, one can query a building block about its _expected utility improvement_ (EUI) via

$$\delta \leftarrow \texttt{get\_eui}(B_{g,D}). \tag{8}$$

Note that, different from EU, EUI is the expected _improvement_ over the current observed utility if given K more budget units. While sharing some similarity with EI in BO, EUI works on the level of optimization process (building blocks), while EI in BO is implemented for one single iteration in BO. In VolcanoML, we estimate EUI by taking the mean of the observed improvements from history, following Levine et al [50].

3.3 Three Types of Building Blocks

Decomposition is the cornerstone of VolcanoML's design. Given a search space, apart from exploring it jointly, there are two classical ways of decomposition — to partition the search space via conditioning on different values of a certain variable (in a similar spirit of _variable elimination_ [15]), or to decompose the problem into multiple smaller ones by introducing equality constraints (in a similar spirit of _dual decomposition_ [11]). This inspires VolcanoML's design, which supports three types of building blocks: (1) _joint block_ that simply optimizes the input subspace using Bayesian optimization; (2) _conditioning block_ that further divides the input subspace into smaller ones by conditioning on one particular input variable; and (3) _alternating block_ that partitions the input subspace into two and optimizes each one _alternately_. Note that both _conditioning block_ and _alternating block_ would generate new building blocks with smaller subgoals. We next present the implementation details for each type of building block.

3.3.1 Joint Block

A joint block directly optimizes its subgoal via Bayesian optimization (BO) [77]. Specifically, BO based method - SMAC [32] has been used by many applications where evaluating the objective function is computationally expensive. It constructs a probabilistic surrogate model M to capture the relationship between the input variables $\bar{\boldsymbol{x}}$ (i.e., hyperparamters in AutoML) and the objective function value ψ (e.g., the validation loss), and this surrogate model is utilized to suggest a new promising configuration to evaluate for each iteration. It then refines M iteratively using past evaluation observations $(\bar{\boldsymbol{x}}, \psi)$.

Based on the BO framework, the implementation of `do_next!` for a joint block consists of the following three steps:

1. Use the surrogate model M to select a configuration $\bar{\boldsymbol{x}}$ that maximizes an acquisition function. In our implementation, we use _expected improvement_ (EI) [38] as the acquisition function, which has been widely used in BO community.
2. Evaluate the selected configuration $\bar{\boldsymbol{x}}$ and obtain its result about the objective function $f_g(\bar{\boldsymbol{x}})$ (i.e., the subgoal). Due to the randomness of most ML algorithms, we assume that $f(\boldsymbol{x})$ cannot be observed directly but rather through noisy observation $\psi = f_g(\bar{\boldsymbol{x}}) + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2)$, where $\mathcal{N}$ is the normal distribution.
3. Refit the surrogate model M on the observed $(\bar{\boldsymbol{x}}, \psi)$.

Algorithm 1: The do_next! of conditioning block

Input: A conditioning block $B_{g,D}$, times to play each arm L, total budget K.

1 Let $B_1, ..., B_m$ be all active (have not been eliminated) child blocks;
2 **for** $1 \leq i \leq L$ **do**
3   **for** $1 \leq j \leq m$ **do**
4   do_next!(B_j);
5 **for** $1 \leq j \leq m$ **do**
6      $[l_j, u_j] \leftarrow$ get_eu(B_j, K);
7 Eliminate child blocks that are *dominated* by others, using $[l_j, u_j]$ for $1 \leq j \leq m$;

Algorithm 2: The init of alternating block

Input: An alternating block $B_{g,D}$ with search space $\bar{x} = \bar{y} \cup \bar{z}$.

1 Initialize $\bar{y}$ and $\bar{z}$ with default values $\bar{y}_0$ and $\bar{z}_0$;
2 $B_1 \leftarrow$ init$(f, \bar{z}, \bar{z}_0, D)$;
3 $B_2 \leftarrow$ init$(f, \bar{y}, \bar{y}_0, D)$;
4 **for** $1 \leq i \leq L$ **do**
5   do_next(B_1);
6      $\bar{y}_i \leftarrow$ get_current_best(B_1);
7   set_var$(B_2, \bar{y}, \bar{y}_i)$;
8   do_next(B_2);
9      $\bar{z}_i \leftarrow$ get_current_best(B_2);
10   set_var$(B_1, \bar{z}, \bar{z}_i)$;

Early-Stopping based Optimization. For large datasets, early-stopping based methods, e.g., Successive Halving [36], Hyperband [51], BOHB [21], MFES-HB [57], etc, can terminate the evaluations of poorly-performing configurations in advance, thus speeding up the evaluations. VOLCANOML supports MFES-HB [57], which combines the benefits of Hyperband and Multi-fidelity BO [90,81], to optimize a joint block, in addition to vanilla BO.

3.3.2 Conditioning Block

A conditioning block decomposes its input $\bar{x}$ into $\bar{x} = \{x_c\} \cup \bar{y}$, where x_c is a single variable with domain $\mathbb{D}_{x_c}$. It then creates one new building block for each possible value $d \in \mathbb{D}_{x_c}$ of x_c:

$$\min_{\bar{y}} g_d(\bar{y}; D) \equiv f(\{x_c = d, \bar{y}\}; D). \tag{9}$$

As a result, $|\mathbb{D}_{x_c}|$ new (child) building blocks are created.

The conditioning block aims to identify optimal value for x_c, and many previous AutoML researchers have used Bandit algorithms for this purpose [63,36, 53,57]. In VOLCANOML, we follow these previous work and model it as a multi-armed bandit (MAB) problem, while our framework is flexible enough to incorporate other algorithms when they are available. There are $|\mathbb{D}_{x_c}|$ arms, where each arm corresponds to a child block. Playing an arm means invoking the do_next! primitive of the corresponding child block.

Algorithm 1 illustrates the implementation of do_next! for a conditioning block. It starts by playing each arm L times in a Round-Robin fashion (lines 2 to 4). Here, L is a user-specified configuration parameter of VOLCANOML. In our current implementation, we set $L = 5$. We then obtain the lower and upper bounds of the expected utility of each child block by invoking its get_eu primitive (lines 5 to 6), and eliminate child blocks that are dominated by others (line 7).

The elimination works as follows. Consider two blocks B_i and B_j: if the upper bound u_i of B_i is less than the lower bound l_j of B_j, then the block B_i is eliminated. An eliminated arm/block will not be played in future invocations of do_next!.

Remark: We have simplified the above elimination criterion by using the lower and upper bounds calculated given K budget units for *each arm*. In fact, these K budget units are *shared* by all the arms, and as a result, each arm actually has fewer budget units than K. Our assumption is that, K is sufficiently large so that one can play *all arms* until (the observed distribution of rewards of) *each arm* converges. Otherwise, the lower and upper bounds obtained may be *over-optimistic*, and as a result, may lead to incorrect eliminations. Fortunately, our assumption usually holds in practice, where arms converge relatively fast.

3.3.3 Alternating Block

An alternating block decomposes its input search space into $\bar{x} = \bar{y} \cup \bar{z}$, and explores $\bar{y}$ and $\bar{z}$ in an *alternating* way. Similarly, we also model the optimization in alternating block as an MAB problem. Algorithm 2 illustrates how its init primitive works. It first creates two child blocks B_1 and B_2, which will focus on optimizing for $\bar{y}$ and $\bar{z}$ respectively (lines 1 to 3). It then (again) views B_1 and B_2 as two arms and plays them using Round-Robin (lines 4 to 10). Note that, when B_1 optimizes $\bar{y}$ (resp. when B_2 optimizes $\bar{z}$), it uses the current best $\bar{z}$ found by B_2 (resp. the current best $\bar{y}$ found by B_1). This is done by the set_var primitive (invoked at line 7 for B_2 and line 10 for B_1).

One problem of our alternating MAB formulation is that the utility improvements of the two building blocks often vary dramatically in practice. For example, some applications are very sensitive to the features being used (e.g., normalized vs. non-normalized features) while hyper-parameter tuning will offer little or even no improvement. In this case, we should spend more

---

Algorithm 3: The `do_next!` of alternating block

Input: An alternating block $B_{g,D}$ with budget K.

1   $\delta_1 \leftarrow$ `get_eui`(B_1);
2   $\delta_2 \leftarrow$ `get_eui`(B_2);
3   **if** $\delta_1 \geq \delta_2$ **then**
4      $\bar{z}_{\text{best}} \leftarrow$ `get_current_best`(B_2);
5   `set_var`($B_1, \bar{z}, \bar{z}_{\text{best}}$);
6   `do_next`(B_1);
7   **else**
8      $\bar{y}_{\text{best}} \leftarrow$ `get_current_best`(B_1);
9   `set_var`($B_2, \bar{y}, \bar{y}_{\text{best}}$);
10   `do_next`(B_2);

---

resources on looking for good features instead of tuning hyper-parameters. Our key observation is that, the *expected utility improvement* (EUI) decays as optimization proceeds. As a result, we propose to use EUI as an indicator that measures the *potential* of pulling an arm further. Algorithm 3 illustrates the details of this idea when used to implement the `do_next!` primitive.

Specifically, Algorithm 3 starts by polling the EUI of both child blocks (lines 1 and 2). Recall that the EUI is estimated by taking the mean of historic observations. It then compares the EUIs and picks the arm/block with larger EUI to play next (lines 3 to 10). Before pulling the winner arm, again it will use the current best settings found by the other arm/block (lines 4 to 6, lines 8 to 10).

3.3.4 Discussion: Pros and Cons of Building Blocks

While the joint block is the most straightforward way to solve the optimization problem associated, it is difficult to scale Bayesian optimization to a large search space [88,53]. The alternating block addresses this scalability issue by decomposing the search space into two smaller subspaces, though with the assumption that the improvements of the two subspaces are *conditionally independent* of each other. As a result, the alternating block is a better choice when such an assumption approximately holds. The advantage of alternating block with this assumption can solve the optimization problem efficiently by decomposing the huge and joint search space into two smaller subspaces (efficiency). While the issue behind this lies in that the alternating block cannot converge to the optimal solution (effectiveness) when the two subspaces are highly dependent. We expect this assumption approximately hold; if not, the alternating block still has its position when dealing with the "efficiency vs. effectiveness" trade-off when the search space is large. The conditioning block is capable of pruning the search space *as optimization proceeds*, when bad arms are pulled less often or will

not be played anymore, with the limitation that it can only work for conditional variables that are *categorical*. For non-categorical variables, one possible way to use conditioning blocks is to split the value range of variables. For example, given a numerical variable that ranges from 1 to 3, we split it into two ranges, which are [1, 2) and [2, 3). During the optimization iteration, we first choose one sub-range and then optimize the splitted space along with its corresponding subspace.

In addition, VOLCANOML uses bandit-based algorithms from the existing literature [50,53] as default in both the alternating and conditioning block, and other bandit-based algorithms, such as successive halving [36], Hyperband [51], BOHB [21] and MFES-HB [57], can also be used in these blocks.

3.3.5 Discussion: Comparing Different Building Blocks

Joint blocks are the default blocks that can be applied to all problems. When the search space is rather large, conditioning and alternating blocks can be helpful. If the search space contains a categorical hyper-parameter, under which the subspace of each choice is conditionally independent with each other, the conditioning block can be used instead of exploring the entire space. If the search space can be decomposed into two approximately independent subspaces, the alternating block can be applied to this case. As a result, a scalable system needs to be able to decompose the problem in different ways and pick the most suitable building blocks. This forms a VOLCANOML execution plan, which we will describe in the next section. In Section 4, we explore the possibility of automatically choosing building blocks to use by maximizing the empirical accuracy of different execution plans, given a pre-defined set of datasets.

3.3.6 Discussion: Continue Tuning in Conditional Block

As introduced in Section 3.3.2, in the conditional block of VOLCANOML, we store the lower and upper bounds of the expected utility of each child block. VOLCANOML eliminates those potentially bad blocks based on the two bounds. When new algorithms are added into the search space, we extend the previously survived candidate algorithm set in the conditional block with those new algorithms and play each candidate in a round-robin fashion as described in Section 3.3.2. After VOLCANOML evaluates those new algorithms with sufficient budget, the conditional block follows the bandit algorithm and eliminates bad candidates with low upper bounds from the candidate set. This process still follows Algorithm 1, only with a difference that the child
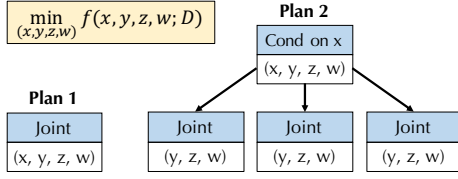
Fig. 3 Two different execution plans for the same optimization problem. Each plan corresponds to a different way to decompose the same search space (x, y, z, w).

Fig. 4 VolcanoML's execution plan for the same search space as explored by `auto-sklearn`. Here 'Alg' and 'HP' correspond to Algorithm and hyper-parameters respectively.

blocks are extended with new algorithms. Therefore, it's quite natural and easy to support continue tuning in VolcanoML.

4 VolcanoML Execution Plan

Given a pre-defined search space, the input of VolcanoML is (1) a dataset D, (2) a utility metric (e.g., cross-validation accuracy) which defines the objective function f, and (3) a time budget. VolcanoML then decomposes a large search space into an *execution plan*, following some specific *decomposition strategy*.

4.1 Execution Plan

VolcanoML *Execution Plan*. Due to the space limitation, we omit the formal definition of a VolcanoML execution plan. A VolcanoML execution plan is a *tree* of building blocks. The root node corresponds to a building block solving the problem f with the entire search space, which can be further decomposed into multiple sub-problems. For each generated sub-problem, a building block (from the three candidates) is applied to solve the corresponding problem. In addition, all the leaf nodes must be the joint blocks. Since joint block does not decompose the search space, it can not be in any paths from the root node to leaf node. As an example, Figure 3 illustrates two possible execution plans for $f(x, y, z, w; D)$. **Plan 1** contains only a single root building block as a joint block, whereas **Plan 2** first introduces a conditioning block on x, and then creates one lower level of building blocks for each possible value of x (in Figure 3, we assume that $|\mathbb{D}_x| = 3$).

VolcanoML *Execution Model*. To execute a VolcanoML execution plan, we follow a Volcano-style execution that is similar to a relational database [28] — the system invokes the `do_next!` of the root node, which then invokes the `do_next!` of one of its child nodes, propagating until the leaf node. At any time, one can invoke the `get_current_best` of the root node, which returns the current best solution for the entire search space.
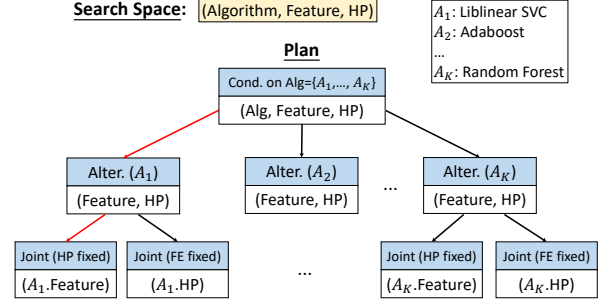
VolcanoML *Plan for* `auto-sklearn`. Figure 4 presents a VolcanoML execution plan for the same search space explored by `auto-sklearn`, which consists of the joint search of *algorithms*, *features transformations operators*, and *hyper-parameters*. Instead of conducting the search process in a single joint block, as was done by `auto-sklearn`, VolcanoML first decomposes the search space via a conditioning block on *algorithms* — this introduces a MAB problem in which each arm corresponds to one particular algorithm. It then decomposes each of the conditioned subspaces via an alternating block between feature engineering and hyper-parameter tuning. The whole subspace of feature engineering (resp. that of hyper-parameter tuning) is optimized by a joint block. Note that this execution plan is similar to the regular plan of human experts, in which experts usually try different algorithms and optimize the feature engineering operations and hyper-parameters alternatively for specific well-performing algorithms.

Concretely, Figure 4 shows a search space for AutoML with K choices of ML algorithms. During each iteration, starting from the root node, VolcanoML selects the child node to optimize until it reaches a leaf node and then optimizes over the subspace in the leaf node. As shown by the red lines in Figure 4, in this iteration, VolcanoML only tunes the feature engineering pipeline of algorithm A_1 while fixing its algorithm hyper-parameters.

Alternative Execution Plans. Note that the execution plan in Figure 4 is not the only possible one. Our flexible and scalable framework in VolcanoML allows us to explore different execution plans before reaching the proposed one, and in the next section 4.2 we introduce the way of automatic plan generation. The reason why we choose this plan is due to the fundamental property of the AutoML search space — we observe that, the optimal choices of *features* are different across *algorithms*, which implies that we can first decompose the
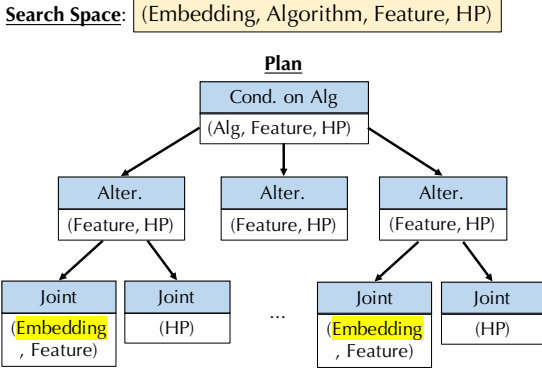
Search Space: (Embedding, Algorithm, Feature, HP)

Plan

Fig. 5 VOLCANOML's execution plan for a larger search space enriched by an additional embedding selection stage.

search space along *ML algorithms*. The improvements introduced by feature engineering and hyper-parameter tuning are largely complementary (See A.1.2 for more details), and thus we can optimize them *alternately*. For feature engineering (resp. hyper-parameter tuning), the subspace is small enough to be handled by a single joint block efficiently. In Section 4.2, we will list the possible plans of the coarse-grained level and further discuss the opportunity of automatic plan generation.

VOLCANOML *Plan for Enriched Search Space.* We can easily extend VOLCANOML and enable functionalities that are not supported by most AutoML systems. For example, Figure 5 illustrates an execution plan for a search space with an additional stage — *embedding selection*. Given an input, e.g., image or text, we first choose embeddings based on a collection of TensorFlow Hub pre-trained models and then conduct algorithm selection, feature engineering, and hyper-parameter tuning. We use an execution plan as illustrated in Figure 5, having the embedding selection step jointly optimized together with the feature engineering.

4.2 Automatic Plan Generation

In principle, the design of VOLCANOML opens up the opportunity for automatic plan generation — given a collection of benchmark datasets, one could automatically search for the best decomposition strategy of the search space and come up with a physical plan automatically. While the complete treatment of this problem is beyond the scope of this paper, we illustrate the possibility with a straightforward strategy. We automatically enumerate all possible execution plans in a coarse-grained level and find that our manually specified execution plan in Figure 4 outperforms the alternatives. The five execution plans are as follows:

- Plan 1 - J(Joint). Optimize over the entire space using a joint block.
- Plan 2 - C(Conditioning). Use a conditioning block on the choice of machine learning algorithms, and then optimize each subspace using joint blocks.
- Plan 3 - A(Alternating). Use an alternating block to separate the entire space into feature engineering space and combined algorithm selection and hyperparameter tuning (CASH) space.
- Plan 4 - AC(Alternating then Conditioning). Use an alternating block to separate the entire space into feature engineering and CASH space, and then use a conditioning block on the choice of algorithm.
- Plan 5 - CA(Conditioning then Alternating). Use a conditioning block on the choice of machine learning algorithms, and then optimize the subspace of feature engineering and algorithm hyperparameters alternately. See Plan 5 in Figure 6 for more details.
- TPOT - `TPOT`. In essence, the execution plan of `TPOT` also uses a single joint block. The difference between `TPOT` and Plan 1 is that `TPOT` uses the evolutionary algorithm while Plan 1 uses the Bayesian optimization.
- AUSK - `autosklearn`. The execution plan of `autosklearn` also uses a single joint block. The difference between `autosklearn` and Plan 1 is their ensemble strategy. Concretely, `autosklearn` build the ensemble model over all the evaluated models while Plan 1 builds it over a fixed number of well-performed models as VOLCANOML does.

Indeed, automatic plan generation can find the optimal solutions with techniques like reinforcement learning. However, one critical problem behind the automatic plan generation is the overhead introduced by constructing and searching for a new execution plan. Here, generating execution plans may take a massive amount of training cost, and automatic plan generation may involve building and evaluating an extremely large volume of plans. Moreover, if the user only has a limited budget, automated plan generation can easily run out of the budget while not providing a decent execution plan.

It is still an open question of whether we can support finer-grained partition of the search space (e.g., different plans for different subspace of features), and moreover, whether we can conduct efficient automatic plan optimization without enumerating all possible plans. These are exciting future directions, and we expect the endeavor to be non-trivial. We hope that this paper sets the ground for this line of research in the future (e.g., rule-based heuristics or reinforcement learning).

Further Discussion. We abstract a VolcanoML execution plan as a tree of building blocks. The root
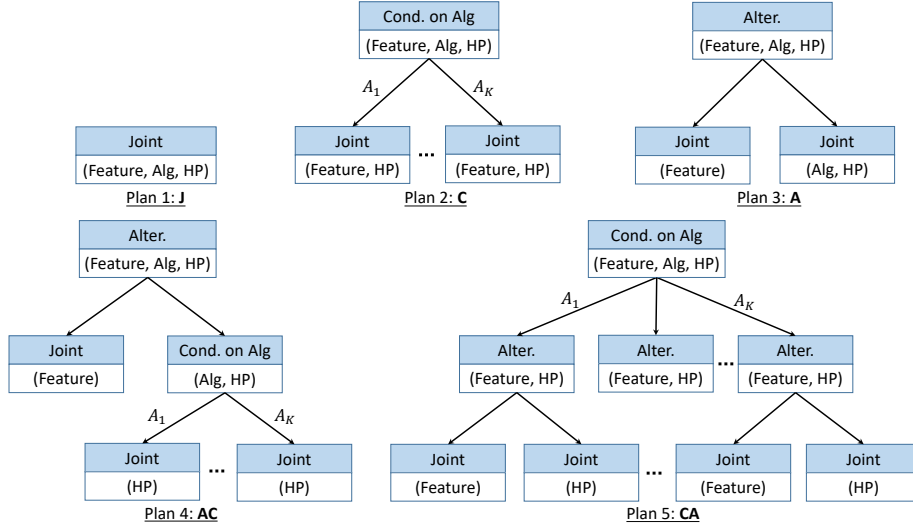
Fig. 6 Five execution plans in the task granularity.

node corresponds to a building block solving the problem with the entire search space, which can be further decomposed into multiple building blocks if necessary. Three kinds of building blocks can be used to build the tree-structured execution plan. Reinforcement Learning (RL) could be a straightforward solution to generate execution plans automatically. The key decisions involve how to define the states, rewards, and actions. We can define the current state by encoding the current structure of the tree and the optimization problem to decompose. When all leaf nodes in the tree are joint blocks, we can execute the current decomposition plan. And we can take the validation accuracy as the reward. Each action corresponds to apply a decomposition strategy to an optimization problem by adding a building block to the tree's some leaf node. The RL agent builds and evaluates each execution plan iteratively by trying different actions. The goal of the agent is to find the plan that achieves the optimal evaluation result.

4.3 Progressive Optimization Methods

Unlike the optimization strategy used in Figure 4, the progressive methods [62] can optimize the search space in a top-down manner. Take the default tree-structured space (Plan 5) shown in Figure 6 as an example, a progressive method first tries different algorithms in the conditional block while keeping all other hyperparameters by default. After evaluating all algorithm candidates, it fixes the best algorithm and enters the search space under this algorithm. Then, it optimizes the space of feature engineering while keeping the algorithm hyperparameters by default. Finally, by fixing the best

found feature engineering operators, it optimizes the algorithm hyperparameters and obtains the final configuration. The main advantage of progressive methods is that, they enjoy high efficiency in exploring the space because they only need to optimize the blocks following a path from the root to the leaves. However, they also have two weak points: (1) While the best algorithm is chosen by keeping other hyperparameters by default, there is a risk that the algorithm found progressively may not be the optimal one; (2) Only one algorithm is explored in the optimization process, and it leads to a lack of diversity in the model pool for the final ensemble. The original optimization strategy deals with the weak points by applying the bandit-based algorithm. It evaluates each algorithm by trying different combinations of other hyperparameters so that it can further compute the expected utility of each algorithm. Meanwhile, since all algorithms are evaluated for some given budget, the evaluation history is diverse, which helps generate a better model ensemble.

5 Further Optimization with Meta-learning

One class of optimizations that we support is *meta-learning* [84,87] — given previous runs of the system over similar workloads, to transfer the knowledge and better help the workload at hand. Depending on the type of different building blocks, we support different meta-learning strategies.

5.1 Meta-learning for Conditioning Blocks

For conditioning block on variable x, it introduces a multi-armed bandit problem with $|\mathbb{D}_x|$ arms, and its objective is to identify the optimal arm. One natural meta-learning strategy is to learn, given the dataset D, a much smaller subset of arms $A \subseteq \mathbb{D}_x$ that includes the optimal arm. This could explicitly reduce the search space in the conditioning block from $\mathbb{D}_x$ to A. We use a meta-learning strategy based on RankNet [10].

During the training process, we are given a training history over multiple previous datasets $D_1, ..., D_n$. We are given the relative relationships between different arms on different datasets

$$\mathcal{T} = \{(A_j, A_k, D_i) : A_j, A_k \in \mathbb{D}_x\}, \tag{10}$$

where $(A_j, A_k, D_i) \in \mathcal{T}$ means that A_j performs better than A_k on dataset D_i. We are also given a meta-feature extractor h_D for dataset and a meta-feature extractor h_A for arms. Both types of extractors will map a dataset (resp. an arm) to an m-dimensional real-valued vector. The model that we are trying to learn is a multi-layer perceptron (MLP) model taking as input a dataset embedding and an arm embedding, with the following learning objective:

$$\min_{\Theta} \sum_{(D_i, A_j, A_k) \in \mathcal{T}} l_+ \left(\sigma(r_j^{(i)} - r_k^{(i)}) \right) + l_- \left(\sigma(r_k^{(i)} - r_j^{(i)}) \right)$$
$$\text{where} \quad r_j^{(i)} = MLP(h_D(D_i), h_A(A_j); \Theta),$$
$$r_k^{(i)} = MLP(h_D(D_i), h_A(A_k); \Theta), \tag{11}$$

where σ is the sigmoid function, l_+ is the hinge loss with positive label, and l_- is the hinge loss with negative label.

During inference, the MLP with parameter Θ takes the vector that consists of h_D and h_A as input, and outputs a score. The best subset of arms can then be selected based on these scores.

5.2 Meta-learning for Joint Blocks

A joint block uses BO method that can be slow when the underlying search space is large. An intuitive optimization is to leverage BO history $H_1 = \{(x_j^1, y_j^1)\}_{j=1}^{n^1}, ..., H_n$ from n previous datasets $D_1, ..., D_n$. This motivates the meta-learning based BO that can speed up the convergence of search in the current joint block.

When executing joint block on a new dataset, we are given the historical observations $H_1, ..., H_n$ from n previous datasets on the same search space, and the observations in the current task is H_T. We use a scalable meta-learning method, RGPE [23], to accelerate BO. First, for each previous dataset D_i, we train a Gaussian process model M_i on the corresponding observations from H_i. Then we build a surrogate model M_{meta} to guide the search in this joint block, instead of the original surrogate M_T fitted on H_T only. The prediction of M_{meta} at point $\boldsymbol{x}$ is given by

$$y \sim \mathcal{N}(\sum_i w_i \mu_i(\boldsymbol{x}), \sum_i w_i \sigma_i^2(\boldsymbol{x})), \tag{12}$$

where w_i is the weight of base surrogate M_i, and μ_i and σ_i^2 are the predictive mean and variance from base surrogate M_i. The weight w_i reflects the similarity between the previous task and current task. Therefore, M_{meta} carries the knowledge of search on previous tasks, which can greatly accelerate the convergence of the search in current joint block. We then use the following ranking loss function L, i.e., the number of misranked pairs, to measure the similarity between previous tasks and current task:

$$L(M_i, H_T) = \sum_{j=1}^{n^T} \sum_{k=1}^{n^T} \mathbb{1}((M_i(\boldsymbol{x}_j) <_i (\boldsymbol{x}_k) \oplus (y_j < y_k)), \tag{13}$$

where $\oplus$ is the exclusive-or operator, $n_T = |H_T|$, $\boldsymbol{x}_j$ and y_j are the sample point and its performance in H^T, and $M_i(\boldsymbol{x}_j)$ means the prediction of M_i on point $\boldsymbol{x}_j$. Based on the ranking loss function, the weight w_i is set to the probability that M_i has the smallest ranking loss on H_T, that is, $w_i = \mathbb{P}(i = \text{argmin}_j L(M_j, H_T))$. This probability can be estimated using MCMC sampling.

Example. When applied to end-to-end AutoML, the joint block is used to select configurations from a joint search space, e.g., the search of hyper-parameter configurations or features given a specific ML algorithm. Although the optimal configuration may be different across tasks, the performance surface of configurations in current task may be similar to some in previous tasks due to the relevancy between tasks. In this case, BO history on previous datasets can be utilized to guide the configuration search via the above meta-learning based BO method.

6 Experimental Evaluation

We compare VOLCANOML with state-of-the-art AutoML systems. In our evaluation, we focus on three perspectives: (1) the *performance* of VOLCANOML given

the *same* search space explored by existing systems, (2) the *scalability* of VOLCANOML given larger search spaces, and (3) the *extensibity* of VOLCANOML to integrate new components into the search space of AutoML pipelines.

6.1 Experimental Setup

AutoML Systems. We evaluate VOLCANOML as well as two open-source AutoML systems: `auto-sklearn` [22] and `TPOT` [69]. In addition, we also compare VOLCANOML with four commercial AutoML platforms from Google, Amazon AWS, Microsoft Azure, and Oracle. Both VOLCANOML and `auto-sklearn` support meta-learning, while `TPOT` does not. For fair comparison with `TPOT`, we also use VOLCANOML$^-$ and AUSK$^-$ to denote the versions of VOLCANOML and `auto-sklearn` when meta-learning is disabled. Our implementation of VOLCANOML is available at `https://github.com/PKU-DAIR/mindware`.

Datasets. To compare VOLCANOML with academic baselines, we use 60 real-world ML datasets from the OpenML repository [85], including 40 for classification (CLS) tasks and 20 for regression (REG) tasks. 10 of the 40 classification datasets are relatively large, each with 20k to 110k data samples; the other 30 are of medium size, each with 1k to 12k samples. In addition, we also use datasets from six Kaggle competitions (See Table 3 for details) to compare VOLCANOML with four commercial platforms.

AutoML Tasks. We define three kinds of real-world AutoML tasks, including (1) a general classification task on 30 medium datasets, (2) a general regression task on 20 medium datasets, and (3) a large-scale classification task on 10 large datasets.

To test the scalability of the participating systems, we design three search spaces that include 20, 29, and 100 hyper-parameters, where the smaller search space is a subset of the larger one. We run VOLCANOML and the baseline AutoML systems against each of the three search spaces. The time budget is 900 seconds for the smallest search space and 1,800 seconds for the other two, when performing the general classification task (1); the time budget is increased to 5,400 and 86,400 seconds respectively, when performing the general regression task (2) and the large-scale classification task (3).

Utility Metrics. Following [22], we adopt the metric *balanced accuracy* for all classification tasks — compared with standard (classification) accuracy, it assigns equal weights to classes and takes the average of classwise accuracy. For regression tasks, we use the *mean squared error* (MSE) as the metric.

In our evaluation, we repeat each experiment 10 times and report the average utility metric. In each experiment, we use four fifths of the data samples in each dataset to search for the best ML pipeline and report the utility metric on the remaining fifth.

Methodology for Comparing AutoML Systems. To compare the overall test result of each AutoML system on a wide range of datasets, we use the *average rank* as the metric following [3]. For each dataset, we rank all participant systems based on the result of the best ML pipeline they have found so far; we then take the average of their ranks across different datasets. In addition, we use statistical testing to determine ties and adjust the rankings [16].

Training Data for Meta-learning. The results for meta-learning are obtained from running Bayesian optimization on 90 classification datasets and 50 regression datasets collected from OpenML. For classification, we collect the results by optimizing the balanced accuracy, accuracy, f1 score and AUC. For regression, we collect the results by optimizing the mean squared error, mean absolute error and r2 value. When VOLCANOML receives a new task and the optimization target is one of the above metrics, VOLCANOML will use all the evaluation results with this metric to train the RankNet in the conditional block and RGPE in the joint block. In our experiments, to ensure the current task does not occur in the results for meta-learning, we apply the leave-one-out strategy. For example, when we optimize Dataset A, we will use all other results except A for meta-learning.

More Details. We include the details of search space and programming API in Appendix A.2, experiment datasets in Appendix A.3.

6.2 End-to-End Comparison

We first evaluate the participant AutoML systems given the search space explored by `auto-sklearn`. Figure 7 presents the results of VOLCANOML compared to `auto-sklearn` (AUSK) and `TPOT` on the 30 classification tasks and the 20 regression tasks, respectively. For classification tasks, we plot the classification accuracy improvement (%); for regression tasks, we plot the *relative MSE improvement* Δ, which is defined as $\Delta(m_1, m_2) = \frac{s(m_2) - s(m_1)}{\max(s(m_2), s(m_1))}$, where $s(\cdot)$ is MSE on the test set. Overall, VOLCANOML outperforms `auto-sklearn` and `TPOT` on 25 and 23 of the 30 classification tasks, and on 17 and 15 of the 20 regression tasks, respectively.

We also conduct experiments to evaluate VOLCANOML with different *time budgets*. Figure 8 presents the results on four large classification datasets. We

(a) VOLCANOML vs. AUSK on CLS

(b) VOLCANOML vs. AUSK on REG

(c) VOLCANOML vs. TPOT on CLS

(d) VOLCANOML vs. TPOT on REG

Fig. 7 End-to-End results on 30 OpenML classification (CLS) datasets and 20 OpenML regression (REG) datasets.

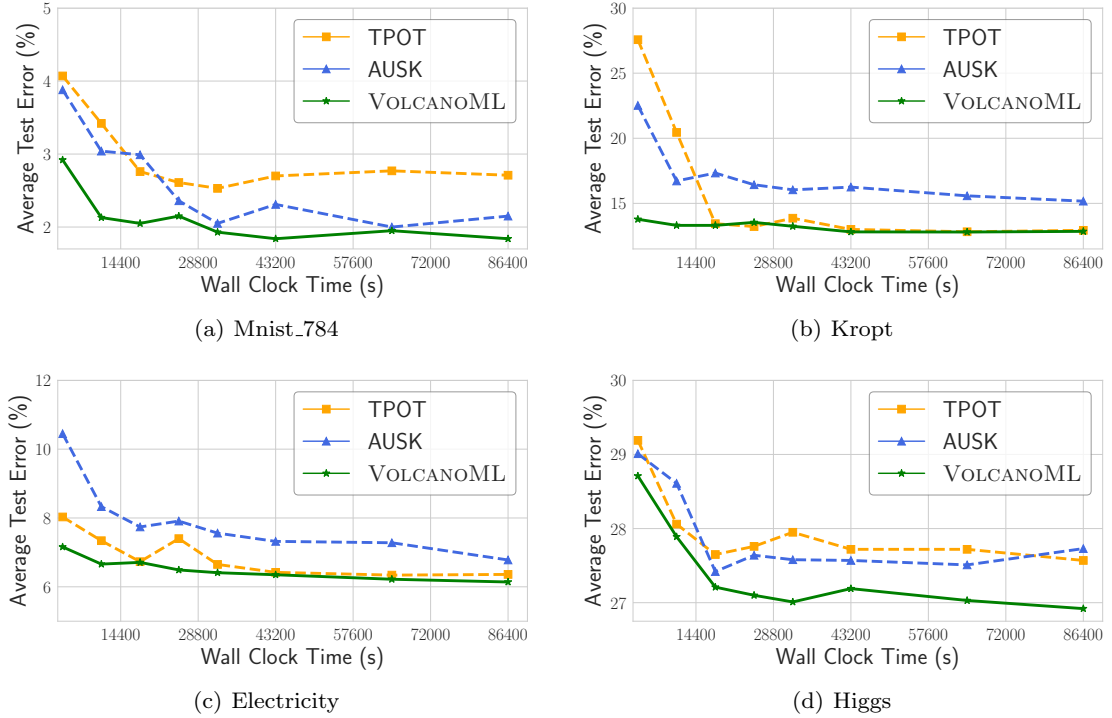(a) Mnist_784

(b) Kropt

(c) Electricity

(d) Higgs

Fig. 8 Average test errors on four large datasets with different time budgets.

Table 1 Average ranks on 30 classification (CLS) datasets and 20 regression (REG) datasets with three different search spaces. (The lower is the better)

Search Space - Task	TPOT	AUSK⁻	AUSK	VolcanoML⁻	VolcanoML
Small - CLS	3.09	3.07	3.01	2.94	**2.89**
Medium - CLS	3.2	3.32	3.27	2.78	**2.43**
Large - CLS	3.29	3.77	3.57	2.72	**1.65**
Small - REG	**2.98**	3.02	3.0	3.02	**2.98**
Medium - REG	2.95	3.3	3.12	**2.75**	2.88
Large - REG	3.1	3.85	3.82	2.15	**2.08**

observe that VolcanoML exhibits consistent performance over different time budgets. Notably, on `Higgs`, VolcanoML achieves 27.2% test error within 4 hours, which is better than the performance of the other two systems given 24 hours. Furthermore, we also design additional experiments to evaluate the consistency of system performance given different (larger) time budgets and search spaces, and more details can be found in the following sections.

We further study the scalability of the participant systems on the three aforementioned search spaces. Without meta-learning, VolcanoML achieves the best average rank for both the classification and regression tasks — on the small search space (with 20 hyper-parameters), VolcanoML performs slightly better than `auto-sklearn` and `TPOT`, and it performs significantly better on the medium (with 29 hyper-parameters) and large (with 100 hyper-parameters) search spaces. For meta-learning, we present more empirical results and discussions in Section 6.6.

6.3 Search Space Enrichment

We now evaluate the *extensibility* of VolcanoML via two experiments with *enriched* search spaces.

Adding `Data_Balancing` Operator. In the first experiment, we implement the feature engineering operator "`smote_balancer`" – a popular over-sampling techinique proposed for the overfitting problem, and incorporate it into the aforementioned *balancing* stage of feature engineering (FE) (Section 3.1). Note that `auto-sklearn` cannot support this fine-grained enrichment of the search space. Table 2 presents the results of `auto-sklearn`, VolcanoML without enrichment, and VolcanoML with enrichment, on five imbalanced datasets. We observe that enriching the search space brings further improvement, e.g., VolcanoML with enrichment outperforms `auto-sklearn` by 3.57% (balanced accuracy) on the dataset `pc2`.

Supporting Embedding Selection. In the second experiment, we add a new stage "embedding selection" into the FE pipeline, with two candidate embedding-extraction operators (i.e., two pre-trained models). This

allows VolcanoML to deal with images, which is difficult for `auto-sklearn` and `TPOT` to support using existing code. We implement two pre-trained models to generate embeddings for images, and we evaluate VolcanoML with the enriched search space on the Kaggle dataset `dogs-vs-cats`. We observe that VolcanoML achieves 96.5% test accuracy, which is significantly better than 70.4% obtained by VolcanoML without considering embeddings.

Table 2 Test accuracy (%) of VolcanoML with and without the enrichment of "`smote_balancer`" operator.

Dataset	AUSK	VolcanoML⁻	VolcanoML
sick	97.29	97.31	**97.34**
pc2	86.70	86.91	**90.27**
abalone	66.86	65.97	**67.32**
page-blocks(2)	94.70	95.29	**96.69**
hypothyroid(2)	99.62	99.64	**99.64**

6.4 Comparison with 4 Industrial Platforms

We run additional experiments on six Kaggle competitions (See Table 3 for dataset statistics) over four commercial baselines (AutoML services from Google, AWS, Azure and Oracle) as follows:

- Google Cloud AutoML on unknown running environment (not transparent to users).
- AWS Sagemaker AutoPilot on an instance 'ml.m5.4xlarge' with 16 Intel Xeon® Platinum 8175M processors and 64G memory.
- Azure Automated ML on two instances 'STANDARD_D12' with totally 8 unknown processors and 56G memory.
- Oracle Data Science on an instance 'VM.Standard2.2' with 2 2.0 GHz Intel® Xeon® Platinum 8167M processors and 30G memory.
- VolcanoML on an Ali-cloud instance 'ecs.hfc6.2xlarge' with 8 3.10 GHz Intel® Xeon® Platinum 8269CY processors and 30G memory.
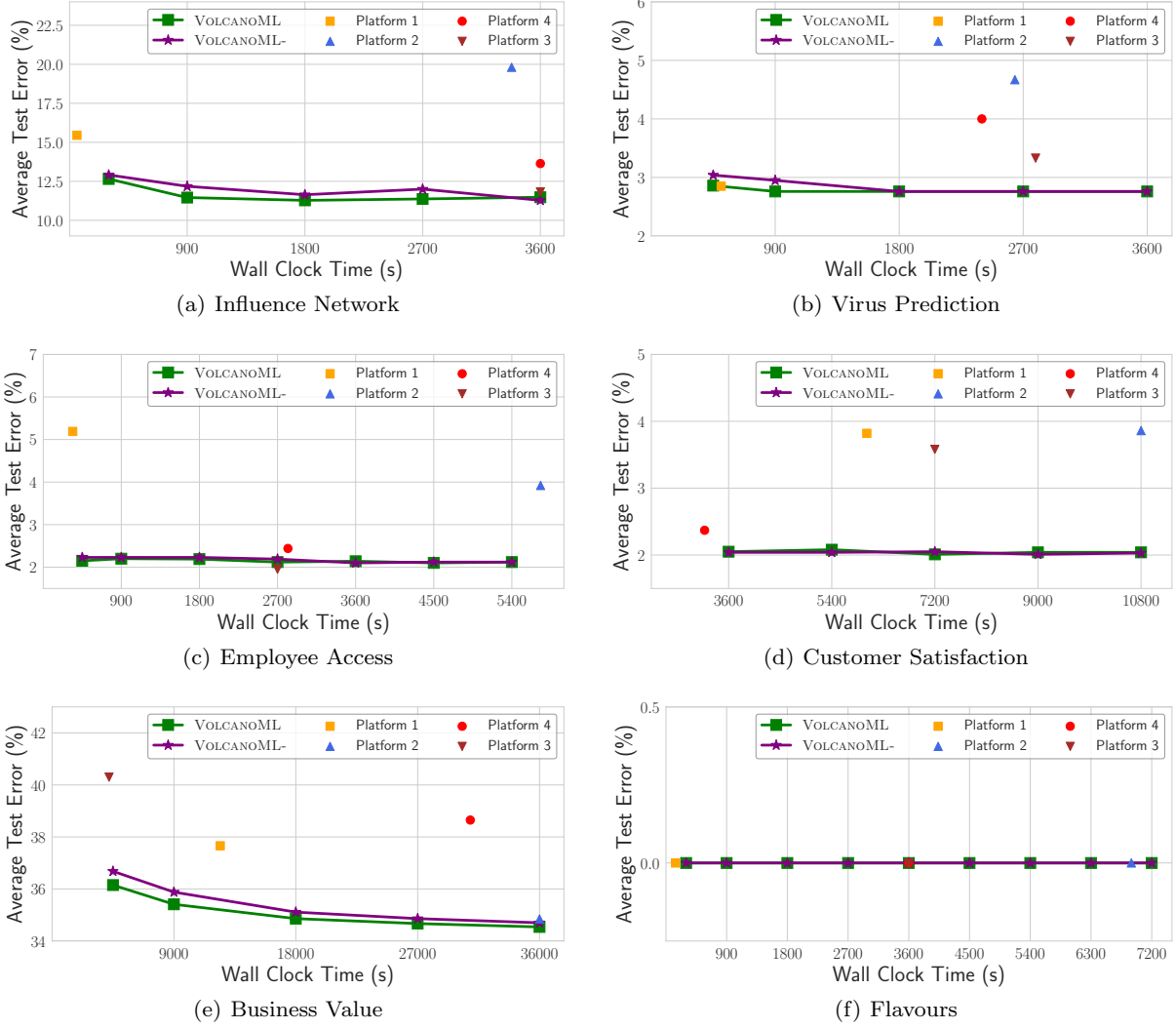
Fig. 9 Test error on Kaggle competitions compared with commericial baselines.

Table 3 Kaggle dataset information.

Datasets	Classes	Samples	Features
Influencers in Social Networks	2	5500	22
West-Nile Virus Prediction	2	10506	11
Employee Access Challenge	2	32769	9
Santander Customer Satisfaction	2	76020	369
Predicting Red Hat Business Value	2	2197291	12
Flavors of Physics	2	38012	49

Due to the different design principles (different hardware and parallelism) of commercial manufacturers, it is very hard to set up exactly the same environment settings. We set the the maximal time budget as 10 hours and use cost as an additional metric. Here, we anonymously refer to these platforms as Platform 1-4. Figure 9 show the results of VolcanoML and the platforms. We observe that VolcanoML- (without meta-learning) achieves satisfactory results com-

pared with those cloud solutions on the six tasks. Due to the large initial search space, VolcanoML- performs slightly worse than VolcanoML (with meta-learning) in the beginning on Influence Network, Virus Prediction, and Business Value. Given more time budget (i.e., fix the x-axis to some time budget), VolcanoML and VolcanoML- show similar results and often outperform the considered commercial platforms. This demonstrates VolcanoML's effectiveness against the commercial AutoML baselines.

6.5 Scalability on Different Search Space

To evaluate the scalability of each system, we design three search spaces of different sizes. The small search space only contains four feature selectors (*select percentile, select generic univariate, extra trees preprocess-*

Table 4 Average ranks on 30 classification (CLS) datasets and 20 regression (REG) datasets with three different search spaces (The lower is the better). The budget is 1800 seconds for classification and 5400 seconds for regression.

Search Space - Task	TPOT	AUSK	VolcanoML
Small - CLS	2.03	1.98	1.98
Medium - CLS	1.95	2.21	**1.83**
Large - CLS	1.97	2.43	**1.60**
Small - REG	2.00	2.00	2.00
Medium - REG	2.05	2.30	**1.65**
Large - REG	2.10	2.20	**1.70**

Table 5 Average ranks on 30 classification (CLS) datasets and 20 regression (REG) datasets with three different search spaces (The lower is the better). The budget is 3600 seconds for classification and 10800 seconds for regression.

Search Space - Task	TPOT	AUSK	VolcanoML
Small - CLS	2.03	1.97	2.0
Medium - CLS	1.90	2.27	**1.83**
Large - CLS	2.03	2.53	**1.43**
Small - REG	1.95	2.00	2.05
Medium - REG	1.95	2.30	**1.75**
Large - REG	1.98	2.28	**1.75**

Table 6 Average ranks on 30 classification (CLS) datasets and 20 regression (REG) datasets with three different search spaces (The lower is the better). The budget is 7200 seconds for classification and 21600 seconds for regression.

Search Space - Task	TPOT	AUSK	VolcanoML
Small - CLS	2.00	2.00	2.00
Medium - CLS	1.97	2.23	**1.80**
Large - CLS	1.92	2.40	**1.68**
Small - REG	2.00	2.00	2.00
Medium - REG	1.95	2.23	**1.83**
Large - REG	2.00	2.10	**1.90**

ing, and *liblinear SVM preprocessing*) and uses *random forest* as the ML algorithm. The medium search space contains the same four feature selectors as the small one and uses *linear_svc(r)*, *random forest*, and *AdaBoost* as the ML algorithms. The large search space is the entire search space described in Section 3.1. The three spaces include 20, 29, and 100 hyper-parameters, respectively, and the smaller space is a subset of the larger one.

To further investigate the result of VolcanoML over 1) different time budgets and 2) different search spaces, we conducted additional experiments to run each system given 1800 / 5400 seconds, 3600 / 10800 seconds and 7200 / 21600 seconds for classification / regression tasks over the small, medium and large search spaces respectively. These numbers are chosen by following the settings in papers of auto-sklearn and TPOT. The experiments include 50 AutoML tasks (30 for classification and 20 for regression), and we use the metric — average rank to measure each system. Tables 4, 5 and 6 show the results over three different search spaces given different time budgets. We can observe that, with the increase of time budget and search space, Vol-
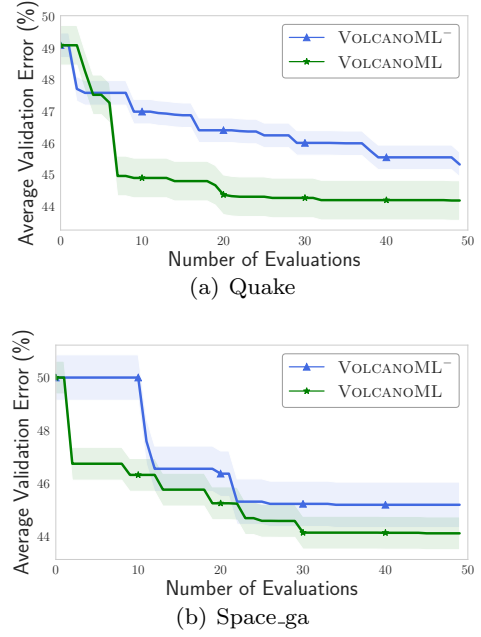

(a) Quake

(b) Space_ga

Fig. 10 The result of the first 50 evaluations on 'quake' and 'space_ga' using LibSVM. VolcanoML$^-$ refers to VolcanoML with meta-learning based BO disabled.

canoML still achieves the best average rank, and performs better compared with auto-sklearn and TPOT.

6.6 Results about Meta-Learning based Optimization

Meta-learning in Joint Block. Figure 10 shows the improvement of meta-learning in a joint block. Compared with VolcanoML$^-$, the validation error drops significantly in the first 10 evaluations on VolcanoML, which indicates that meta-learning captures the information of the historical tasks and performs an effective warm-start. When achieving the same validation error as vanilla VolcanoML$^-$, VolcanoML reduces the number of evaluations by eight-fold on quake and two-fold on space_ga.

Meta-learning in Conditioning Block. To compare the performance of RankNet, we also used *Light-GBM* to model the relationship between algorithm performance and tasks by transforming the ranking problem into a binary classification problem. The input to both *LightGBM* and RankNet is the same. We adopted 10-fold validation mechanism to evaluate each method; the meta-learner is learned on the training set, and validated on the validation set. In addition, we measured the performance of each method using the metric '*mAP@5*', which is the mean Average Precision to predict the top-5 algorithms. RankNet and *Light-GBM* gets 0.87 and 0.62 mAP@5 score respectively.

This demonstrates the more powerful expressiveness of neural networks (RankNet) than traditional ML algorithms (*LightGBM*).

Table 1 also summarizes the performance of meta-learning in terms of the average ranks. With meta-learning, the average rank of VolcanoML is dramatically improved compared with `auto-sklearn`. Overall, VolcanoML with meta-learning achieves the best result over large search space.

6.7 Evaluations on Different Execution Plans

To address the inefficiency of automated plan generation, we apply the execution plan that performs well on most tasks. Since there are lots of ways to decompose AutoML space, enumerating and evaluating each execution plan is impossible. To avoid enumerating the entire search space, we list all the execution plans (a small plan set) in the coarse-grained level — sub-task, and this small plan set covers both the frameworks in existing AutoML systems and the strategies used by human experts. Concretely, we can obtain this small plan set by decomposing the search space according to the sub-tasks in AutoML: feature engineering, hyperparameter tuning and algorithm selection. There are in total five execution plans (See Figure 6) — *J, C, A, AC, and CA*. If we decompose the search space in a more fine-grained level, the plan set will be larger. Note that, most of the existing open-source AutoML systems, e.g., `auto-sklearn` and `TPOT` correspond to the execution plan — plan 1 (*J*). We evaluate each of them on 20 classification tasks and 10 regression tasks, and the results can be found in Tables 7 and 8. We can have that the proposed execution plan used in VolcanoML (i.e., plan 5 - *CA*) outperforms the other alternatives on most tasks (a smaller average rank). This demonstrates the effectiveness of the proposed execution plan over the potential competitors.

Furthermore, if we look at the performance of the existing open-source AutoML systems, i.e., `auto-sklearn` and `TPOT`, which fall into the execution plan 1 - *J*. As shown in Tables 7 and 8, we find that the execution plan used in VolcanoML (i.e., *CA*) outperforms the two AutoML systems on most tasks (a smaller average rank).

The test results are shown in Tables 7 and 8. We can have that the best execution plan varies over datasets. Surprisingly, we find that Plan 5, which is the execution plan introduced in VolcanoML, achieves the first place on 16 of the total 30 tasks with an average rank of 2.45 (the lower, the better). `TPOT` and `autosklearn` that use a single joint block achieves an average rank of 3.83 and 4.98 respectively. Therefore, the proposed

execution plan (Plan 5) in VolcanoML sets up a very competitive AutoML baseline for our further research on VolcanoML, e.g., automatic plan generation.

6.8 Additional Experiment Results

Comparison with early-stopping methods. Table 9 show the results for VolcanoML compared with early-stopping based methods on five classification datasets and five regression datasets in Table 9. The datasets are of medium size, each of which contains 8192 samples. The settings follow Section 6.1, and the compared baselines are Hyperband [51], BOHB [21], and MFES-HB [57]. Remind that VolcanoML uses SMAC [32] in the joint block by default. While the execution plan in VolcanoML is independent of optimization algorithms, we also implement VolcanoML +, which applies the MFES-HB algorithm in the joint block. From Table 9, we observe that VolcanoML with SMAC outperforms the three early-stopping methods, and the performance is further improved when we combine the benefits of both VolcanoML execution plans and early-stopping optimization methods.

Results on large datasets. Table 10 shows the results on ten large datasets with a budget of 18,000 seconds. VolcanoML is the best on eight of them. Figure 11 shows the validation errors on four of those datasets. When achieving the same validation error compared with `TPOT` and `auto-sklearn`, VolcanoML obtains a speed-up of 4.3-10.5× and 4.8-11×, respectively.

Continue tuning in conditional block. To show the process of continue tuning, we present a case study on the dataset pc4. We add three algorithms (LightGBM, Extra Trees, and Liblinear SVC) after tuning 7 other algorithms in VolcanoML for 1200 seconds. The total budget is 1800 seconds. The trend of the number of active blocks is plotted in Figure 12. When new algorithms come, VolcanoML with restarting re-optimizes the extended search space, and it takes another 540s to reduce the number of active algorithms to 6. For VolcanoML with continue tuning, the number of active algorithms is 4 (1 survived + 3 added) when new algorithms are added, and it takes another 220 seconds to reduce the number to 1, which is LightGBM in the added algorithms. As continue tuning avoids exploring the search space of those eliminated algorithms, VolcanoML with continue tuning improves the test accuracy on pc4 to 86.44% compared with 84.74% achieved by VolcanoML with restarting.

Comparison with progressive methods. We also compare the progressive strategies with original ones on

Table 7 Test accuracy with different execution plans for classification.

Dataset	Plan 1	Plan 2	Plan 3	Plan 4	Plan 5	TPOT	AUSK
puma8NH	0.8275	0.8312	0.8271	0.8280	0.8303	0.8306	**0.8325**
kin8nm	0.8808	0.8886	0.8886	0.8654	**0.8910**	0.8706	0.8834
cpu_cmall	0.9122	**0.9126**	0.9126	0.9027	__0.9127__	0.9121	0.9106
puma32H	0.8849	0.8864	0.8848	0.8835	0.8894	**0.8955**	0.8830
cpu_act	0.9303	__0.9315__	0.9305	0.9302	0.9309	__0.9312__	0.9298
bank32nh	0.7896	0.7889	0.7838	0.7891	__0.7957__	0.7593	__0.7957__
mc1	0.8796	0.8904	0.8722	0.8721	**0.8975**	0.8835	0.8896
delta_elevators	0.8763	0.8760	0.8779	0.8766	0.8790	**0.8835**	0.8743
jm1	0.6718	0.6721	0.6581	0.6473	0.6692	0.6415	**0.6772**
pendigits	0.9932	0.9936	0.9929	**0.9945**	0.9937	0.9931	__0.9944__
delta_ailerons	0.9235	0.9240	0.9242	0.9225	0.9259	**0.9278**	0.9259
wind	0.8587	0.8589	0.8566	0.8583	**0.8593**	0.8542	0.8494
satimage	0.8961	0.8954	0.8965	0.8946	**0.8981**	0.8961	0.8793
optdigits	0.9889	0.9889	0.9883	0.9889	0.9889	**0.9902**	0.9818
phoneme	0.8799	0.8832	0.8808	0.8791	**0.8866**	0.8812	0.8770
spambase	0.9401	**0.9406**	0.9379	0.9387	0.9386	0.9385	0.9358
abalone	0.6688	0.6679	0.6618	0.6614	0.6680	__0.6748__	__0.6751__
mammography	0.8740	0.8783	0.8577	0.8755	**0.8787**	0.8568	0.8762
waveform	0.8948	0.8961	0.8900	0.8835	0.8952	0.8955	**0.9040**
pollen	0.4934	__0.5013__	0.5012	__0.5013__	__0.5013__	0.4961	0.4896
Average Rank	4.30	2.98	4.80	5.13	**2.58**	3.83	4.40

Table 8 Test mean square error with different execution plans for regression.

Dataset	Plan 1	Plan 2	Plan 3	Plan 4	Plan 5	TPOT	AUSK
bank8FM	**0.0008**	**0.0008**	**0.0008**	**0.0008**	**0.0008**	**0.0008**	0.0009
bank32nh	0.0071	__0.0069__	0.0070	0.0071	__0.0069__	__0.0069__	0.0070
kin8nm	0.0067	0.0068	0.0073	0.0076	**0.0066**	0.0092	0.0148
puma8NH	10.3020	**10.0822**	10.1293	10.1091	10.1698	10.1043	10.2109
cpu_small	7.3994	7.0854	7.0069	7.1741	**7.0051**	7.4058	8.7286
wind	8.9650	8.9636	8.8993	9.2930	**8.6976**	8.8618	9.2261
cpu_act	5.0067	4.8762	4.7950	4.7983	**4.7790**	4.8373	6.4232
puma32H	0.0001	0.0001	0.0001	0.0001	**0.0000**	0.0001	0.0001
sulfur	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	0.0003	0.0003
space_ga	0.0115	**0.0093**	0.0098	0.0099	0.0098	0.0098	0.0108
Average Rank	4.95	3.00	3.40	4.30	**2.20**	4.00	6.15

(a) Classification

Dataset (ID)	VolcanoML	VolcanoML +	HyperBand	BOHB	MFES-HB
puma8NH (816)	83.03	**83.12**	83.01	82.91	82.96
kin8nm (807)	89.10	**89.14**	88.28	88.70	89.12
cpu_small (735)	91.27	**91.33**	90.97	91.08	91.14
puma32H (752)	89.55	89.61	89.34	89.43	**89.73**
cpu_act (761)	**93.12**	93.01	92.88	92.96	92.97
Average Rank	2.2	**1.4**	4.6	4.2	2.6

(b) Regression

Dataset (ID)	VolcanoML	VolcanoML +	HyperBand	BOHB	MFES-HB
puma8NH (225)	10.1698	**10.1642**	10.1843	10.1654	10.2619
kin8nm (189)	**0.0066**	0.0069	0.0081	0.0073	0.0072
cpu_small (227)	**7.0051**	7.1341	7.4657	7.5272	7.5363
puma32H (308)	__0.0000__	__0.0000__	__0.0000__	__0.0000__	__0.0000__
cpu_act (573)	**4.7790**	4.7524	4.8778	5.2856	5.1506
Average Rank	2.0	**1.8**	3.6	3.6	4.0

Table 9 Test accuracy (%) and test mean squared error of VolcanoML compared with early-stopping methods. VolcanoML + refers to the combination of VolcanoML with MFES-HB.

Table 10 Test balanced accuracy on 10 large datasets.

Datasets	TPOT	AUSK	VolcanoML
mnist_784	0.9724	0.9701	**0.9795**
letter(2)	__0.9969__	0.9939	__0.9969__
kropt	0.8656	0.8267	**0.8669**
mv	__0.9997__	0.9994	__0.9997__
a9a	0.8129	**0.8250**	0.8215
covertype	0.7124	0.7098	**0.7152**
2dplanes	0.9291	**0.9297**	0.9293
higgs	0.7235	0.7258	**0.7279**
electricity	__0.9327__	0.9226	__0.9329__
fried	__0.9296__	0.9280	**0.9300**

result, we apply it as the original strategy by default for VolcanoML.

7 Conclusion

five classification tasks and five regression tasks. The settings follow Section 6.1 and the results are shown in Table 11. We observe that the original strategy outperforms the progressive one on 8 of the 10 tasks. As a

In this paper, we have presented VolcanoML, a scalable and extensible framework that allows users

(a) Wind
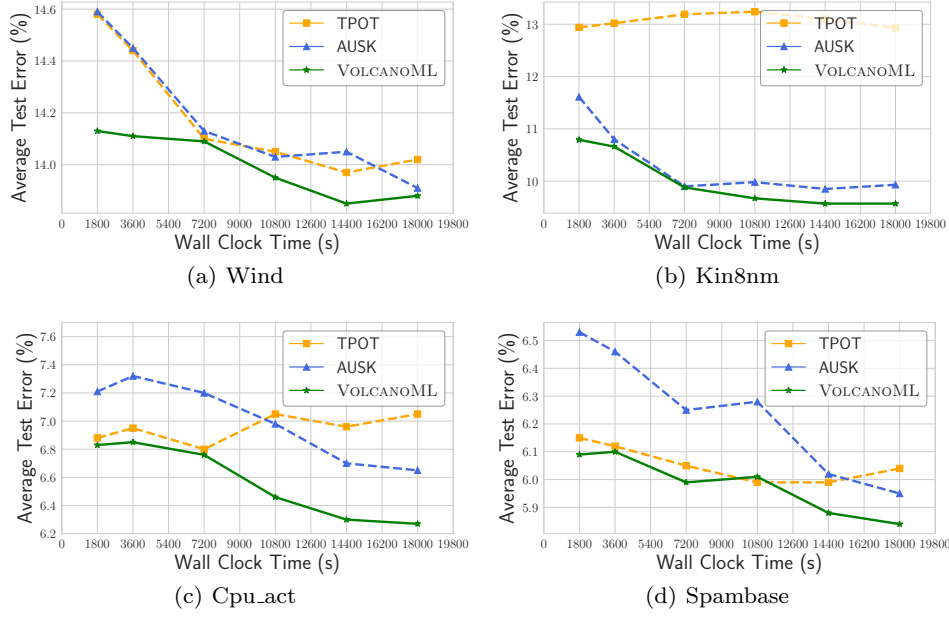

(b) Kin8nm

(c) Cpu_act

(d) Spambase

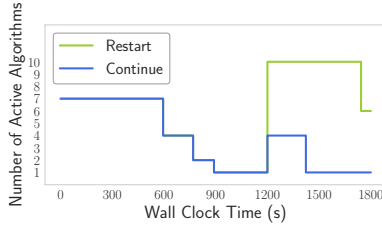**Fig. 11** Test errors on four medium datasets given different time budgets.

Fig. 12 The trend of the number of active algorithms in the conditional block on pc4.

(a) Classification			(b) Regression		
Dataset (ID)	Original	Progressive	Dataset (ID)	Original	Progressive
puma8NH (816)	**83.03**	82.99	puma8NH (225)	**10.1698**	10.2437
kin8nm (807)	**89.10**	88.72	kin8nm (189)	0.0066	**0.0065**
cpu_small (735)	**91.27**	**91.27**	cpu_small (227)	**7.0051**	7.2181
puma32H (752)	**89.55**	88.97	puma32H (308)	**0.0000**	0.0001
cpu_act (761)	**93.12**	93.09	cpu_act (573)	**4.7790**	4.8321

Table 11 Test accuracy (%) and test mean squared error for two optimization strategies on classification and regression tasks.

to design decomposition strategies for large AutoML search spaces in an expressive and flexible manner. VOLCANOML introduces novel building blocks akin to relational operators in database systems that enable expressing search space decomposition strategies in a *structured* fashion – similar to relational execution plans. Moreover, VOLCANOML introduces a Volcano-style execution model, inspired by its classic counterpart that has been widely used for relational query evaluation, to execute the decomposition strategies it yields. Experimental evaluation demonstrates that VOLCANOML can generate more efficient decomposi-

tion strategies that also lead to performance-wise better ML pipelines, compared to state-of-the-art AutoML systems.

A Appendix

In this section, we describe more details about the background, system design and implementations.

A.1 AutoML Formulations and Motivations

A.1.1 Formulations

Definition and Notation. There are K candidate algorithms $\mathcal{A} = \{A^1, ..., A^K\}$. Each algorithm A^i has a corresponding hyper-parameter space Λ_i. The algorithm A^i with hyper-parameter configuration λ and new feature set F is denoted by $A^i_{(\lambda, F)}$. Given the dataset $D = \{D_{train}, D_{valid}\}$ of a learning problem, the AutoML problem is to find the joint algorithm, feature, and hyper-parameter configuration
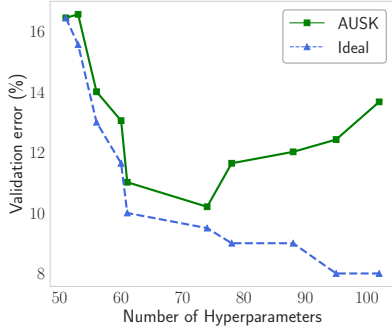
Fig. 13 Validation error on `pc4` when increasing the number of hyper-parameters in `auto-sklearn` given the same time budget.
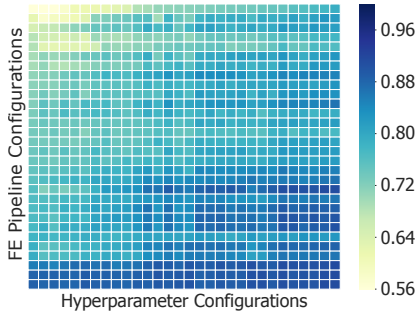

Fig. 14 The performance distribution of ML pipelines constructed by 30 FE and HPO configurations on `fri_c1` using Random Forest. For FE configurations, the performance increases from top to down; for HPO configurations, the performance increases from left to right (The deeper, the better).

$A^*_{(\lambda^*, F^*)}$ that minimizes the loss metric (e.g., the validation error on D_{valid}):

$$A^*_{(\lambda^*, F^*)} = \operatorname*{argmin}_{A^i \in \mathcal{A}, \lambda \in \Lambda^i, F \in \mathcal{F}^i} \mathcal{L}(A^i_{(\lambda, F)}; D), \qquad (14)$$

where $\mathcal{F}^i = Gen(A^i, D, \mathbf{op})$ is the feature space of A^i that can be generated from the raw feature (data) set D, and $\mathbf{op}$ is the set of available FE operators.

Challenge: Ever-growing Search Space. Enriching the search space can lead to performance improvement since the enriched search space may bring better configurations. However, an ever-growing search space can significantly increase the complexity of searching for ML pipelines. Existing AutoML systems usually can only explore very limited configurations in a huge search space, and thus suffer from the low-efficiency issue [53] that hampers the effectiveness of AutoML systems. In Figure 13, we provide a brief example of `auto-sklearn`, one state-of-the-art system AutoML system. Its search algorithm cannot scale to a high-dimensional search space [53]. To alleviate this issue, in this paper we focus on developing a scalable AutoML system.

A.1.2 Observations and Motivations about AutoML

We now present several important observations that inspired the design of VOLCANOML.

Observation 1. The search space can be partitioned according to ML algorithms. The entire search space is the union of the search spaces of individual algorithms, i.e., $\Omega = \{S^1, ..., S^K\}$, where S^i is the joint space of features and hyper-parameters, i.e., $S^i = (\Lambda^i \times \mathcal{F}^i)$.

Observation 2. The sub-space of algorithm A^i can be very large, e.g., in `auto-sklearn`, S^i usually includes more than 50 hyper-parameters. When exploring the search spaces via extensive experiments, we observe the following:

- If hyper-parameter configuration λ_1 performs better than λ_2, i.e., $\lambda_1 \leq \lambda_2$, then it often holds that $(\lambda_1, F) \leq (\lambda_2, F)$ for the joint configuration (λ, F) with F fixed;
- If FE pipeline configuration F_1 performs better than F_2, i.e., $F_1 \leq F_2$, then it often holds that $(\lambda, F_1) \leq (\lambda, F_2)$ for the joint configuration (λ, F) with λ fixed.

Figure 14 presents an example for these observations. This motivates us to solve the joint FE and HPO problem via *alternating* optimization. That is, we can alternate between optimizing FE and HPO, and we can fix the FE configuration (resp. HPO configuration) when optimizing for HPO (resp. FE). This alternating manner is indeed similar to how human experts solve the joint optimization problem manually. One obvious advantage of alternating optimization is that each time only a much smaller subspace (Λ^i or $\mathcal{F}^i$) needs to be optimized, instead of the joint space $S^i = (\Lambda^i \times \mathcal{F}^i)$.

Observation 3. The sensitivity of ML algorithms to FE and HPO is often different. Taking Figure 14 for example, compared to HPO, FE has a larger influence on the performance of 'Random Forest' on 'fri_c1'; in this case, optimizing FE *more frequently* can bring more performance improvement.

Observation 4. The above observations motivate the use of *meta-learning*. We can learn (1) the algorithm performance across ML tasks and (2) the configuration selection of each ML algorithm across tasks. Such meta-knowledge obtained from historical tasks can greatly improve the efficiency of ML pipeline search.

Therefore, a scalable AutoML system should include two basic components: (1) an efficient framework that can navigate in a huge search space, and (2) a meta-learning module that can extract knowledge from previously ML tasks and apply it to new tasks.

A.2 VOLCANOML Components and Implementations

A.2.1 Compenents and Search Space

Feature Engineering. The feature engineering pipeline is shown in Figure 2. It comprises four sequential stages: *pre-processors* (compulsory), *scalers* (5 possible operators), *balancers* (1 possible operators) and *feature transformers* (13 possible operstors). For each of the latter three stages, VOLCANOML picks one operator and then execute the entire pipeline. Table 13 presents the details of each operator. The total number of hyper-parameters for FE is 52.

We follow the design of the search space for feature engineering in the existing AutoML systems, e.g., `autosklearn` and `TPOT`. It limits the search space for feature engineering by adopting a fixed pipeline including different stages, and each stage is equipped with an operation (featurizer) that

is selected from a pool of featurizers. The pool of featurizers at each stage is relatively small, and Bayesian optimization can be used to choose the proper featurizers for each stage. When the pool of featurizers is very large, high-dimensional Bayesian optimization algorithms could work better. In many real-world cases, though this architecture is not good enough for feature engineering (effectiveness), there still remains space to explore to conduct feature engineering effectively and efficiently. To support real scenarios, VolcanoML provides API for user-defined feature engineering operators and we recommend users to add domain-specific feature engineering operators to the search space for better search performance. In addition, users can replace the original feature engineering part in VolcanoML with other iterative feature engineering methods easily.

ML Algorithms. VolcanoML implements 11 algorithms for classification and 10 algorithms for regression, with a total of 50 and 49 hyper-parameters respectively. The built-in algorithms include *linear models*, *support vector machine*, *discriminant analysis*, *nearest neighbors*, and *ensembles*. Table 12 presents the details.

Ensemble Methods. Ensembles that combine predictions from multiple base models have been known to outperform individual models, often drastically reducing the variance of the final predictions [17]. VolcanoML provides four ensemble methods: *bagging*, *blending*, *stacking*, and *ensemble selection* [12]. During the search process, the top N_{top} configurations for each algorithm are recorded and the corresponding models are stored. After the optimization budget exhausts, the saved models are treated as the base models for the ensemble method. We use *ensemble selection* as the default method and build an ensemble of size 50.

Table 12 Hyper-parameters of ML algorithms in VolcanoML. We distinguish categorical (cat) hyper-parameters from numerical (cont) ones. The numbers in the brackets are conditional hyper-parameters.

Type of Classifier	#λ	cat (cond)	cont (cond)
AdaBoost	4	1 (-)	3 (-)
Random forest	5	2 (-)	3 (-)
Extra trees	5	2 (-)	3 (-)
Gradient boosting	7	1 (-)	6 (-)
KNN	2	1 (-)	1 (-)
LDA	4	1 (-)	3 (1)
QDA	1	-	1 (-)
Logistic regression	4	2 (-)	2 (-)
Liblinear SVC	5	2 (2)	3 (-)
LibSVM SVC	7	2 (2)	5 (-)
LightGBM	6	-	6 (-)

Type of Regressor	#λ	cat (cond)	cont (cond)
AdaBoost	4	1 (-)	3 (-)
Random forest	5	2 (-)	3 (-)
Extra trees	5	2 (-)	3 (-)
Gradient boosting	7	1 (-)	6 (-)
KNN	2	1 (-)	1 (-)
Lasso	3	-	3 (-)
Ridge	4	1 (-)	3 (-)
Liblinear SVC	5	2 (2)	3 (-)
LibSVM SVC	8	3 (3)	5 (-)
LightGBM	6	-	6 (-)

Table 13 Hyper-parameters of FE operators in VolcanoML.

Type of Operator	#λ	cat (cond)	cont (cond)
One-hot encoder	0	-	-
Imputer	1	-	1 (-)
Minmax	0	-	-
Normalizer	0	-	-
Quantile	2	1 (-)	1 (-)
Robust	2	-	2 (-)
Standard	0	-	-
Weight Balancer	0	-	-
Cross features	1	-	1 (-)
Fast ICA	4	3 (1)	1 (1)
Feature agglomeration	4	3 (2)	1 (-)
Kernel PCA	5	1 (1)	4 (3)
Rand. kitchen sinks	2	-	2 (-)
LDA decomposer	1	1 (-)	-
Nystroem sampler	5	1 (1)	4 (3)
PCA	2	1 (-)	1 (-)
Polynomial	2	1 (-)	1 (-)
Random trees embed.	5	1 (-)	4 (-)
SVD	1	-	1 (-)
Select percentile	2	1 (-)	1 (-)
Select generic univariate	3	2 (-)	1 (-)
Extra trees preprocessing	5	2 (-)	3 (-)
Linear SVM preprocessing	5	3 (3)	2 (-)

A.2.2 Programming Interface

Consider a tabular dataset of raw values in a CSV file, named `train.csv`, where the last column represents the label. We take a classification task as an example. With VolcanoML, only six lines of code are needed for searching and model evaluation.

```
from ... import DataManager , Classifier
dm = DataManager ()
train_node = dm.load_train ('train.csv')
test_node = dm.load_test ('test.csv')
clf = Classifier (**params).fit(train_node)
predictions = clf.predict(test_node)
```

By calling `load_train` and `load_test`, the *data manager* automatically identifies the type of each feature (continuous, discrete, or categorical), imputes missing values, and converts string-like features to one-hot vectors. By calling `fit`, VolcanoML splits the dataset into folds for training and validation, evaluates various configurations, and generates an ensemble from each individual configuration. For users who need to customize the search process, `Classifier` provides additional parameters to specify:

- `time_limit` controls the total runtime of the search process;
- `include_algorithms` specifies which algorithms are included (if not specified, all built-in algorithms are included);
- `ensemble_method` chooses which ensemble strategy to use;
- `enable_meta` determines whether to use meta-learning to accelerate the search process;
- `metric` specifies the metric used to evaluate the performance of each configuration.

Customized Components. VolcanoML provides APIs to easily enrich the search space, such as the stage in FE

pipeline, FE operators, and ML algorithms. The following is the syntax of defining customized components:

```
from ... import add_classifier
from ... import update_FEPipeline
from ... import BaseModel, BaseOperator

# Add new ML algorithm.
class CustomizedModel(BaseModel):
    def fit(x,y): ...
    def predict(x): ...
    def get_search_space(): ...

add_classifier(CustomizedModel)

# Add new FE operator.
class CustomizedOP(BaseOperator):
    def operate(x): ...
    def get_search_space(): ...

# Customize FE pipeline.
update_FEPipeline(['new_stage', ...],
     {'new_stage': [CustomizedOP],
     ...})
```

It is important to note that, `auto-sklearn` does not support adding a new stage or updating the existing stages in the FE pipeline. In addition, `auto-sklearn` cannot add an operator for any stage (e.g., adding `smote_balancer` to the stage `balancer`), while VOLCANOML supports this.

A.3 Experiment Datasets

In our experiments, we splitted each dataset into five folds. Four are used for training and the remaining one is used for testing. The 60 OpenML datasets used are presented as follows (in the form of "dataset_name (OpenML id)"):

Classification Datasets. kc1 (1067), quake (772), segment (36), ozone-level-8hr (1487), space_ga (737), sick (38), pollen (871), analcatdata_supreme (728), abalone (183), spambase (44), waveform(2) (979), phoneme (1489), page-blocks(2) (1021), optdigits(28), satimage (182), wind (847), delta_ailerons (803), puma8NH (816), kin8nm (807), puma32H (752), cpu_act (761), bank32nh (833), mc1 (1056), delta_elevators (819), jm1 (1053), pendigits (32), mammography (310), ailerons (734), eeg (1471), letter(2) (977), kropt (184), mv (881), fried (901), 2dplanes (727), electricity (151), a9a (A2), mnist_784 (554), higgs (23512), covertype (180).

Regression Datasets. stock (223), socmob (541), Moneyball (41021), insurance (A1), weather_izmir (42369), us_crime (315), debutanizer (23516), space_ga (507), pollen (529), wind (503), bank8FM (572), bank32nh (558), kin8nm (189), puma8NH (225), cpu_act (573), puma32H (308), cpu_small (227), visualizing_soil (668), sulfur (23515), rainfall_bangladesh (41539).

Since the datasets `insurance` and `a9a` are not collected in OpenML, we use A1 and A2 as their OpenML ID instead.

References

1. Aguilar Melgar, L., Dao, D., Gan, S., Gürel, N.M., Hollenstein, N., Jiang, J., Karlaš, B., Lemmin, T., Li, T., Li, Y., et al.: Ease. ml: A lifecycle management system for machine learning. In: Proceedings of the Annual Conference on Innovative Data Systems Research (CIDR), 2021. CIDR (2021)

2. Bai, Y., Li, Y., Shen, Y., Yang, M., Zhang, W., Cui, B.: Autodc: an automatic machine learning framework for disease classification. Bioinformatics (2022)

3. Bardenet, R., Brendel, M., Kégl, B., Sebag, M.: Collaborative hyperparameter tuning. In: International conference on machine learning, pp. 199–207. PMLR (2013)

4. Barnes, J.: Azure machine learning. Microsoft Azure Essentials. 1st ed, Microsoft (2015)

5. Baylor, D., Breck, E., Cheng, H.T., Fiedel, N., Foo, C.Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al.: Tfx: A tensorflow-based production-scale machine learning platform. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1387–1395 (2017)

6. Bergstra, J., Bengio, Y.: Random search for hyperparameter optimization. Journal of Machine Learning Research **13**, 281–305 (2012)

7. Bergstra, J.S., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems, pp. 2546–2554 (2011)

8. Boehm, M., Antonov, I., Baunsgaard, S., Dokter, M., Ginthör, R., Innerebner, K., Klezin, F., Lindstaedt, S., Phani, A., Rath, B., et al.: Systemds: A declarative machine learning system for the end-to-end data science lifecycle. arXiv preprint arXiv:1909.02976 (2019)

9. Breck, E., Polyzotis, N., Roy, S., Whang, S., Zinkevich, M.: Data validation for machine learning. In: MLSys (2019)

10. Burges, C.: From ranknet to lambdarank to lambdamart: An overview. Learning **11** (2010)

11. CarøE, C.C., Schultz, R.: Dual decomposition in stochastic integer programming. Operations Research Letters **24**(1-2), 37–45 (1999)

12. Caruana, R., Niculescu-Mizil, A., Crew, G., Ksikes, A.: Ensemble selection from libraries of models. In: Proceedings, Twenty-First International Conference on Machine Learning, ICML 2004 (2004). DOI 10.1145/1015330.1015432

13. Chen, B., Wu, H., Mo, W., Chattopadhyay, I., Lipson, H.: Autostacker: A compositional evolutionary learning system. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 402–409 (2018)

14. De Sa, C., Ratner, A., Ré, C., Shin, J., Wang, F., Wu, S., Zhang, C.: Deepdive: Declarative knowledge base construction. ACM SIGMOD Record **45**(1), 60–67 (2016)

15. Dechter, R.: Bucket elimination: A unifying framework for probabilistic inference. In: Learning in graphical models, pp. 75–104. Springer (1998)

16. Dewancker, I., McCourt, M., Clark, S., Hayes, P., Johnson, A., Ke, G.: A strategy for ranking optimization methods using multiple criteria. In: Workshop on Automatic Machine Learning, pp. 11–20. PMLR (2016)

17. Dietterich, T.G.: Ensemble methods in machine learning. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2000). DOI 10.1007/3-540-45014-9_1

18. Drori, I., Krishnamurthy, Y., Rampin, R., De, R., Lourenco, P., Ono, J.P., Cho, K., Silva, C., Freire, J.: AlphaD3M: Machine Learning Pipeline Synthesis. AutoML Workshop at ICML (2018)

19. Efimova, V., Filchenkov, A., Shalamov, V.: Fast automated selection of learning algorithm and its hyperparameters by reinforcement learning. In: International Conference on Machine Learning AutoML Workshop (2017)

20. Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., Leyton-Brown, K.: Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In: NIPS workshop on Bayesian Optimization in Theory and Practice, vol. 10, p. 3 (2013)

21. Falkner, S., Klein, A., Hutter, F.: Bohb: Robust and efficient hyperparameter optimization at scale. In: International Conference on Machine Learning, pp. 1437–1446. PMLR (2018)

22. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in neural information processing systems, pp. 2962–2970 (2015)

23. Feurer, M., Letham, B., Bakshy, E.: Scalable meta-learning for bayesian optimization using ranking-weighted gaussian process ensembles. In: AutoML Workshop at ICML (2018)

24. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book, 2 edn. Prentice Hall Press, USA (2008)

25. Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S.: Systemml: Declarative machine learning on mapreduce. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 231–242. IEEE (2011)

26. Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., Sculley, D.: Google vizier: A service for black-box optimization. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1487–1495. ACM (2017)

27. Google: Google prediction api,. `https://developers.google.com/prediction` (2020)

28. Graefe, G.: Volcano—An Extensible and Parallel Query Evaluation System. IEEE Transactions on Knowledge and Data Engineering (1994)

29. He, X., Zhao, K., Chu, X.: Automl: A survey of the state-of-the-art. Knowledge-Based Systems **212**, 106622 (2021)

30. Hu, Y.Q., Yu, Y., Tu, W.W., Yang, Q., Chen, Y., Dai, W.: Multi-fidelity automatic hyper-parameter tuning via transfer series expansion. AAAI (2019)

31. Hutter, F., Hoos, H., Leyton-Brown, K.: An efficient approach for assessing hyperparameter importance. In: 31st International Conference on Machine Learning, ICML 2014 (2014)

32. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, pp. 507–523. Springer (2011)

33. Hutter, F., Kotthoff, L., Vanschoren, J. (eds.): Automated Machine Learning: Methods, Systems, Challenges. Springer (2018). In press, available at http://automl.org/book.

34. Hutter, F., Lücke, J., Schmidt-Thieme, L.: Beyond manual tuning of hyperparameters. KI-Künstliche Intelligenz **29**(4), 329–337 (2015)

35. IBM: Ibmwatson studio autoai. `https://www.ibm.com/cloud/watson-studio/autoai` (2020)

36. Jamieson, K., Talwalkar, A.: Non-stochastic best arm identification and hyperparameter optimization. In: Artificial Intelligence and Statistics, pp. 240–248 (2016)

37. Jiang, H., Shen, Y., Li, Y.: Automated hyperparameter optimization challenge at cikm 2021 analyticcup. arXiv preprint arXiv:2111.00513 (2021)

38. Jones, D.R., Schonlau, M., Welch, W.J.: Efficient global optimization of expensive black-box functions. Journal of Global optimization **13**(4), 455–492 (1998)

39. Kandasamy, K., Dasarathy, G., Schneider, J., Póczos, B.: Multi-fidelity bayesian optimisation with continuous approximations. In: International Conference on Machine Learning, pp. 1799–1808. PMLR (2017)

40. Kanter, J.M., Veeramachaneni, K.: Deep feature synthesis: Towards automating data science endeavors. In: 2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015, pp. 1–10. IEEE (2015)

41. Katz, G., Shin, E.C.R., Song, D.: Explorekit: Automatic feature generation and selection. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 979–984. IEEE (2016)

42. Kaul, A., Maheshwary, S., Pudi, V.: Autolearn—automated feature generation and selection. In: 2017 IEEE International Conference on data mining (ICDM), pp. 217–226. IEEE (2017)

43. Khurana, U., Samulowitz, H., Turaga, D.: Feature engineering for predictive modeling using reinforcement learning. In: 32nd AAAI Conf. Artif. Intell. AAAI 2018 (2018)

44. Khurana, U., Turaga, D., Samulowitz, H., Parthasrathy, S.: Cognito: Automated feature engineering for supervised learning. In: 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW), pp. 1304–1307. IEEE (2016)

45. Klein, A., Falkner, S., Bartels, S., Hennig, P., Hutter, F.: Fast bayesian optimization of machine learning hyperparameters on large datasets. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, pp. 528–536 (2017)

46. Komer, B., Bergstra, J., Eliasmith, C.: Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In: ICML workshop on AutoML, vol. 9. Citeseer (2014)

47. Kraska, T.: Northstar: An interactive data science system. Proceedings of the VLDB Endowment **11**(12), 2150–2164 (2018)

48. Krishnan, S., Wang, J., Wu, E., Franklin, M.J., Goldberg, K.: Activeclean: Interactive data cleaning for statistical modeling. Proceedings of the VLDB Endowment **9**(12), 948–959 (2016)

49. LeDell, E., Poirier, S.: H2o automl: Scalable automatic machine learning. In: Proceedings of the AutoML Workshop at ICML, vol. 2020 (2020)

50. Levine, N., Crammer, K., Mannor, S.: Rotting bandits. In: Advances in NIPS, pp. 3074–3083 (2017)

51. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. Proceedings of the International Conference on Learning Representations pp. 1–48 (2018)

52. Li, T., Zhong, J., Liu, J., Wu, W., Zhang, C.: Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. Proceedings of the VLDB Endowment **11**(5), 607–620 (2018)

53. Li, Y., Jiang, J., Gao, J., Shao, Y., Zhang, C., Cui, B.: Efficient automatic cash via rising bandits. In: AAAI, pp. 4763–4771 (2020)

54. Li, Y., Shen, Y., Jiang, H., Bai, T., Zhang, W., Zhang, C., Cui, B.: Transfer learning based search space design for hyperparameter tuning. Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (2022)

55. Li, Y., Shen, Y., Jiang, H., Zhang, W., Li, J., Liu, J., Zhang, C., Cui, B.: Hyper-tune: Towards efficient hyper-parameter tuning at scale. Proceedings of the VLDB Endowment **15** (2022)

56. Li, Y., Shen, Y., Jiang, H., Zhang, W., Yang, Z., Zhang, C., Cui, B.: Transbo: Hyperparameter optimization via two-phase transfer learning. Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (2022)

57. Li, Y., Shen, Y., Jiang, J., Gao, J., Zhang, C., Cui, B.: Mfes-hb: Efficient hyperband with multi-fidelity quality measurements. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 8491–8500 (2021)

58. Li, Y., Shen, Y., Zhang, W., Chen, Y., Jiang, H., Liu, M., Jiang, J., Gao, J., Wu, W., Yang, Z., et al.: Openbox: A generalized black-box optimization service. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, pp. 3209–3219 (2021)

59. Li, Y., Shen, Y., Zhang, W., Jiang, J., Ding, B., Li, Y., Zhou, J., Yang, Z., Wu, W., Zhang, C., et al.: Volcanoml: Speeding up end-to-end automl via scalable search space decomposition. Proceedings of the VLDB Endowment (2021)

60. Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E., Stoica, I.: Tune: A research platform for distributed model selection and training. arXiv preprint arXiv:1807.05118 (2018)

61. Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., et al.: Elastic machine learning algorithms in amazon sagemaker. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 731–737 (2020)

62. Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.J., Fei-Fei, L., Yuille, A., Huang, J., Murphy, K.: Progressive neural architecture search. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 19–34 (2018)

63. Liu, S., Ram, P., Bouneffouf, D., Bramble, G., Conn, A.R., Samulowitz, H., Gray, A.G.: An admm based framework for automl pipeline configuration pp. 4892–4899 (2020)

64. Mohr, F., Wever, M., Hüllermeier, E.: Ml-plan: Automated machine learning via hierarchical planning. Machine Learning **107**(8), 1495–1515 (2018)

65. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., et al.: Ray: A distributed framework for emerging {AI} applications. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), pp. 561–577 (2018)

66. Nakandala, S., Kumar, A., Papakonstantinou, Y.: Incremental and approximate inference for faster occlusion-based deep cnn explanations. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1589–1606 (2019)

67. Nakandala, S., Zhang, Y., Kumar, A.: Cerebro: A data system for optimized deep learning model selection. Proceedings of the VLDB Endowment **13**(12), 2159–2173 (2020)

68. Nargesian, F., Samulowitz, H., Khurana, U., Khalil, E.B., Turaga, D.S.: Learning feature engineering for classification. In: Ijcai, pp. 2529–2535 (2017)

69. Olson, R.S., Moore, J.H.: Tpot: A tree-based pipeline optimization tool for automating machine learning. In: Automated Machine Learning, pp. 151–160. Springer (2019)

70. Poloczek, M., Wang, J., Frazier, P.: Multi-information source optimization. In: Advances in Neural Information Processing Systems, pp. 4288–4298 (2017)

71. Ratner, A., et al.: Snorkel: Rapid training data creation with weak supervision. PVLDB (2017)

72. Rekatsinas, T., Chu, X., Ilyas, I.F., Ré, C.: Holoclean: Holistic data repairs with probabilistic inference. Proceedings of the VLDB Endowment **10**(11) (2017)

73. Research, M.: Microsoft nni. https://github.com/Microsoft/nni (2020)

74. de Sá, A.G., Pinto, W.J.G., Oliveira, L.O.V., Pappa, G.L.: RECIPE: A grammar-based framework for automatically evolving classification pipelines. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2017)

75. Schawinski, K., et al.: Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. MNRAS Letters (2017)

76. Sen, R., Kandasamy, K., Shakkottai, S.: Noisy blackbox optimization with multi-fidelity queries: A tree search approach. arXiv preprint arXiv:1810.10482 (2018)

77. Shahriari, B., Swersky, K., Wang, Z., Adams, R.P., De Freitas, N.: Taking the human out of the loop: A review of bayesian optimization. Proceedings of the IEEE **104**(1), 148–175 (2015)

78. Smith, M.J., Sala, C., Kanter, J.M., Veeramachaneni, K.: The machine learning bazaar: Harnessing the ml ecosystem for effective system development. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 785–800 (2020)

79. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems, pp. 2951–2959 (2012)

80. Swersky, K., Snoek, J., Adams, R.P.: Multi-task bayesian optimization. In: Advances in neural information processing systems, pp. 2004–2012 (2013)

81. Takeno, S., Fukuoka, H., Tsukada, Y., Koyama, T., Shiga, M., Takeuchi, I., Karasuyama, M.: Multi-fidelity bayesian optimization with max-value entropy search and its parallelization. In: International Conference on Machine Learning, pp. 9334–9345. PMLR (2020)

82. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 847–855 (2013)

83. Van Rijn, J.N., Hutter, F.: Hyperparameter importance across datasets. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2367–2376 (2018)

84. Vanschoren, J.: Meta-learning: A survey. CoRR **abs/1810.03548** (2018). URL http://arxiv.org/abs/1810.03548

85. Vanschoren, J., Van Rijn, J.N., Bischl, B., Torgo, L.: Openml: networked science in machine learning. ACM SIGKDD Explorations Newsletter **15**(2), 49–60 (2014)

86. Vartak, M., et al.: Modeldb: A system for machine learning model management. In: HILDA (2016)

87. Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Artificial Intelligence Review (2002). DOI 10.1023/A:1019956318069

88. Wang, Z., Zoghi, M., Hutter, F., Matheson, D., De Freitas, N.: Bayesian optimization in high dimensions via random embeddings. In: Twenty-Third International Joint Conference on Artificial Intelligence (2013)

89. Wistuba, M., Schilling, N., Schmidt-Thieme, L.: Two-stage transfer surrogate model for automatic hyperparameter optimization. In: Joint European conference on machine learning and knowledge discovery in databases, pp. 199–214. Springer (2016)

90. Wu, J., Toscano-Palmerin, S., Frazier, P.I., Wilson, A.G.: Practical multi-fidelity bayesian optimization for hyperparameter tuning. In: Uncertainty in Artificial Intelligence, pp. 788–798. PMLR (2020)

91. Wu, R., Chaba, S., Sawlani, S., Chu, X., Thirumuruganathan, S.: Zeroer: Entity resolution using zero labeled examples. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 1149–1164 (2020)

92. Wu, W., Flokas, L., Wu, E., Wang, J.: Complaint-driven training data debugging for query 2.0. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 1317–1334 (2020)

93. Yao, Q., Wang, M., Chen, Y., Dai, W., Li, Y.F., Tu, W.W., Yang, Q., Yu, Y.: Taking human out of learning applications: A survey on automated machine learning. arXiv preprint arXiv:1810.13306 (2018)

94. Zaharia, M., et al.: Accelerating the machine learning lifecycle with mlflow. IEEE Data Eng. Bull. (2018)

95. Zhang, X., Chang, Z., Li, Y., Wu, H., Tan, J., Li, F., Cui, B.: Facilitating database tuning with hyper-parameter optimization: A comprehensive experimental evaluation. Proceedings of the VLDB Endowment (2021)

96. Zhang, X., Wu, H., Li, Y., Tan, J., Li, F., Cui, B.: Towards dynamic and safe configuration tuning for cloud databases. Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (2022)

97. Zöller, M.A., Huber, M.F.: Benchmark and survey of automated machine learning frameworks. Journal of artificial intelligence research **70**, 409–472 (2021)