VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition

[Scalable Data Science]

Yang Li†‡, Yu Shen†, Wentao Zhang†, Jiawei Jiang‡, Bolin Ding§, Yaliang Li§
Jingren Zhou§, Zhi Yang†, Wentao Wu*, Ce Zhang‡, Bin Cui†
†EECS, Peking University ‡ETH Zurich §Alibaba Group *Microsoft Research, Redmond

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.

PVLDB Reference Format:
Yang Li, Yu Shen, Wentao Zhang, Jiawei Jiang, Bolin Ding, Yaliang Li, Jingren Zhou, Zhi Yang, Wentao Wu, Ce Zhang, and Bin Cui. VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition. PVLDB, 14(11): XXX-XXX, 2021. doi:10.4778/3476249.3476270

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/PKU-DAIR/mindware.

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, e.g., SystemML [18], SystemDS [5], Snorkel [59], ZeroER [77], TFX [6, 51], “Query 2.0” [78], Krypton [54], Cerebro [55], ModelDB [72], MLFlow [80], DeepDive [81], HoloClean [60], ActiveClean [40], and NorthStar [39]. End-to-end AutoML systems [26, 79, 82] 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 [15], TPOT [57], and hyperopt-sklearn [38], whereas most cloud service providers, e.g., Google, Microsoft, Amazon, Alibaba, 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, 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 [62], 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 newly arXiv’ed models to enrich the “Model Base” [44], or to support Cosine annealing 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.

Many existing systems such as *auto-sklearn* [15] and *TPOT* [57] 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 [45, 50]), 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, as shown above, but only some of them can perform well. Without a structured, high-level abstraction of 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 Technical Contributions.** In this paper, we focus on designing a system, *VolcanoML*, which is scalable to a large search space. 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 [17] (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 a range of additional optimizations (Section 4), *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; (2) using an enriched search space with additional feature engineering operators, *VolcanoML* performs significantly better than *auto-sklearn*; and (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.

**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 [22, 26, 79, 82] 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 [27] and solved jointly [15, 57, 68]. Apart from grid search and random search [3], genetic programming [52, 57] and Bayesian optimization (BO) [4, 13, 25, 64, 65] has become prevailing frameworks for this problem. One challenge of end-to-end AutoML is the staggering huge search space that one has to support and many of these methods suffer from scalability issues. In addition, meta-learning [70] 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 [9, 15, 24, 69] 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* [68], *hyperopt-sklearn* [38] and *auto-sklearn* [15] are the main representatives of BO-based AutoML systems. *auto-sklearn* is one of the most popular open-source framework. *TPOT* [57] and *ML-Plan* [52] use genetic algorithm and hierarchical task networks planning respectively to optimize over the pipeline space, and require discretization of the hyper-parameter space. AlphaD3M [11] 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 [8] focuses on ensembling and cascading to generate complex pipelines, and solves the CASH problem [15] via random search. Furthermore, a growing number of commercial enterprises also export their AutoML...
services to their users, e.g., H2O [41], Microsoft’s Azure Machine Learning [2], Google’s Prediction API [20], Amazon Machine Learning [49] and IBM’s Watson Studio AutoAI [28].

Automating Individual Components. Apart from end-to-end AutoML, many efforts have been devoted to studying sub-problems in AutoML: (1) feature engineering [33–36, 56], (2) algorithm selection [12, 15, 38, 45, 50, 68], and (3) hyper-parameter tuning [4, 14, 23, 25, 29, 31, 37, 43, 47, 58, 63, 65, 66, 76]. Meta-learning methods [16, 19, 74] 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 [61] helps users to automate feature engineering, hyper-parameter tuning, and model compression. Recent work [50] leverages the ADMM optimization framework to decompose the CASH problem [15], and solves two easier sub-problems. Berkeley’s Ray project [53] provides the tune module [48] to support scalable hyper-parameter tuning tasks in a distributed environment. Featuretools [32] 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.

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 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 auto-sklearn. 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. In auto-sklearn, it further defines multiple stages of the feature engineering process: (1) preprocessing, (2) rescaling, (3) balancing, and (4) feature transforming. 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).

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. 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.

For example, auto-sklearn handles the above search space jointly and optimizes it with Bayesian optimization (BO) [64]. 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. One interesting design decision in VOLCANOML is to introduce a structured abstraction to express different decomposition strategies. A decomposition 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:

\[
\min_{x_1, ..., x_n, D} f(x_1, ..., x_n, D),
\]

where \( x_1, ..., x_n \) is a set of \( n \) variables and each of them has domain \( D_{x_i} \) for \( i \in [n] \). Together, these \( n \) variables define a search space \( (x_1, ..., x_n) \in \prod D_{x_i} \), \( D \) corresponds to the input dataset, which is a set of input samples. In our setting, \( f(\cdot) \) is a black-box function that we can only evaluate (but not exploiting the derivative). Given constant \( c \) in the composite domain \( c \in \prod D_{x_i} \), we use the notation

\[
f(((x_1, ..., x_n) = c); D)
\]

as the value of evaluating \( f \) by substituting \( (x_1, ..., x_n) \) with \( c \).

**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: \( x_g \subseteq \{x_1, ..., x_n\} \) as a subset of variables, and \( c_g \in \prod D_{x_g} \), as an assignment in the domain of all variables in \( x_g \). Let \( \hat{x}_g \) be all variables that are not in \( x_g \).
Each subgoal defines a function \( f_g \) over a smaller search space, which is constructed by substituting all variables in \( \bar{x}_g \) with \( \bar{e}_g \):

\[
f_g = f(\bar{x}_g / \bar{e}_g) : z \in \bigcap_{x_i \in x_g} \mathcal{D}_{x_i} \mapsto f(\{\bar{x}_g \neq \bar{e}_g, \bar{x}_g = z\}; D).
\]

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

\[
\min f_g(\bar{x}_g, D).
\]

A building block \( B_{g, D} \) imposes several assumptions on \( g \) and \( D \). First, given an assignment \( \bar{e}_g \) to \( \bar{x}_g \), it is able to evaluate the value of the function \( f_g(\bar{e}_g, D) \). Note that such an evaluation can often be expensive and VolcanoML tries to minimize the number of times that such a function is evaluated. Second, given a dataset \( D \), a building block has the knowledge about how to sub-sample a smaller dataset \( D \subseteq D \) and then conduct evaluations on such a subset \( x \mapsto f_g(x; D) \). Third, we assume that the building block has access to a cost model about the cost of an evaluation at \( x, \bar{c}_g, \bar{D} \).

**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

\[
B_{g, D} \leftarrow \text{init}(f, \bar{x}_g, \bar{e}_g, D),
\]

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

\[
\hat{x} \leftarrow \text{get_current_best}(B_{g, D}).
\]

Furthermore, one can ask \( B_{g, D} \) to iterate once via

\[
\text{do_next!}(B_{g, D}),
\]

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

\[
[I, u] \leftarrow \text{get_eu}(B_{g, D}, K).
\]

By adopting a similar design principle used in the existing AutoML systems [15, 50, 57], 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 [45] 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

\[
\delta \leftarrow \text{get_eui}(B_{g, D}).
\]

Note that, different from EU, EUI is the expected improvement over the current observed utility if given \( K \) more budget units. In VolcanoML, we estimate EUI by taking the mean of the observed improvements from history, following Levine et al [42].

### 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 (or a similar spirit of variable elimination [10]), or to decompose the problem into multiple smaller ones by introducing equality constraints (in a similar spirit of dual decomposition [7]). 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) [64]. Specifically, BO based method - SMAC [25] 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 \( \bar{x} \) and the objective function value \( \bar{y} \), and refines \( M \) iteratively using past observations \((\bar{x}, \bar{y})\).

The implementation of **do_next!** for a joint block consists of the following three steps:

1. Use the surrogate model \( M \) to select \( \bar{x} \) that maximizes an acquisition function. In our implementation, we use expected improvement (EI) [30] as the acquisition function, which has been widely used in BO community.
2. Evaluate the selected \( \bar{x} \) and obtain its value about the objective function (i.e., the subgoal) \( \bar{y} = f_g(\bar{x}) + \epsilon \) with \( \epsilon \sim N(0, \sigma^2) \), where \( N \) is the normal distribution.
3. Refit the surrogate model \( M \) on the observed \((\bar{x}, \bar{y})\).

### Early-Stopping based Optimization

For large datasets, early-stopping based methods, e.g., Successive Halving [29], Hyperband [43], BOHB [14], MFES-HB [47], etc, can terminate the evaluations of poorly-performed configurations in advance, thus speeding up the evaluations. VolcanoML supports MFES-HB [47], which combines the benefits of Hyperband and Multi-fidelity BO [67, 75], to optimize a joint block, in addition to vanilla BO.

#### 3.3.2 Conditioning Block

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

\[
\min_{\bar{d}} g_d(\bar{y}; D) = f(\{\bar{x}_c = d, \bar{y}\}; D).
\]

As a result, \( |D_{x_c}| \) new (child) building blocks are created.

The conditioning block aims to identify optimal value for \( x_c \), and many previous AutoML researches have used Bandit algorithms for this purpose [29, 46, 47, 50]. 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 \(|D_{x_c}| \) 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 \( B_1 \) and \( B_2 \): if the upper bound \( u_i \) of \( B_1 \) is less than the lower bound \( l_j \) of \( B_2 \), then the block \( B_1 \) 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 \( \mathcal{X} = \mathcal{Y} \cup \mathcal{Z} \), and explores \( \mathcal{Y} \) and \( \mathcal{Z} \) 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 \( \mathcal{Y} \) and \( \mathcal{Z} \) 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 \( \mathcal{Y} \) (resp. when \( B_2 \) optimizes \( \mathcal{Z} \)), it uses the current best \( \mathcal{Z} \) found by \( B_2 \) (resp. the current best \( \mathcal{Y} \) 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 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.
Figure 1: Two different execution plans for the same optimization problem. Each plan corresponds to a different way to decompose the same search space \((x, y, z, w)\).

Figure 2: 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.

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.

VOLCANOML Execution Plan. Due to space limitation, we omit the formal definition of a VOLCANOML execution plan. Intuitively, 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 building blocks if necessary, as previously described. As an example, Figure 1 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 1, we assume that \(|D_x| = 3\).

VOLCANOML Execution Model. To execute a VOLCANOML execution plan, we follow a Volcano-style execution that is similar to a relational database [21] — the system invokes the \texttt{do_next!} of the root node, which then invokes the \texttt{do_next!} of one of its child nodes, propagating until the leaf node. At any time, one can invoke the \texttt{get_current_best} of the root node, which returns the current best solution for the entire search space.

VOLCANOML Plan for auto-sklearn. Figure 2 presents a VOLCANOML execution plan for the same search space explored by auto-sklearn, which consists of the joint search of algorithms, features, 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 further 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.

Figure 3: VOLCANOML’s execution plan for a larger search space enriched by an additional embedding selection stage.

Concretely, Figure 2 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 2, 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 2 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. We enumerate five possible plans in a coarse-grained level, and the results show that the proposed plan performs best. 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 along ML algorithms. The improvements introduced by feature engineering and hyper-parameter tuning are largely complementary, 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.

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 3 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 3, having the embedding selection step jointly optimized together with the feature engineering.

Discussion: 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 full treatment of this problem is beyond the scope of this paper, we illustrate the possibility with a very simple strategy. We automatically enumerate all possible execution plans in a coarse-grained level, and find that our manually specified execution plan in Figure 2 outperforms the alternatives. There is still an open question that 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 Optimization with Meta-learning.** VolcanoML supports meta-learning based techniques — given previous runs of the system over similar workloads, to transfer the knowledge and better help the workload at hand — to accelerate the optimization process of building blocks. Appendix contains the details.

5 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 extensibility of VolcanoML to integrate new components into the search space of AutoML pipelines.

5.1 Experimental Setup

**AutoML Systems.** We evaluate VolcanoML as well as two open source AutoML systems: auto-sklearn [15] and TPOT [57]. 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/VolcanoML.

**Datasets.** To compare VolcanoML with academic baselines, we use 60 real-world ML datasets from the OpenML repository [71], 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 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 3,400 and 86,400 seconds respectively, when performing the general regression task (2) and the large-scale classification task (3).

**Utility Metrics.** Following [15], 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 class-wise accuracy. For regression tasks, we use the mean squared error (MSE) as the metric.

![Table 1: Average ranks on 30 classification (CLS) datasets and 20 regression (REG) datasets with three different search spaces. (The lower is the better)](https://github.com/VolcanoML)

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 [1]. 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.

**More Details.** We include the details of search space and programming API, experiment settings and additional experiments, and reproductions in the Appendix.

5.2 End-to-End Comparison

We first evaluate the participant AutoML systems given the search space explored by auto-sklearn. Figure 4 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 $\Delta$, which is defined as $\Delta(m_1, m_2) = \frac{s(m_2) - s(m_1)}{s(m_2)}$, 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 5 presents the results on four large classification datasets. We 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.

We further study the scalability of the participant systems on the three aforementioned search spaces. Table 1 summarizes the results in terms of the average ranks. We have two observations: First, 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. Second, with meta-learning, the average rank of VolcanoML is dramatically improved compared to auto-sklearn. Overall, VolcanoML with meta-learning achieves the best result over large search space. 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 Appendix.
We imple-
ment “smote_balancer” – a new feature engineering op-
erator, 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 are not easily supported by both auto-sklearn and TPOT. We implement two pre-trained models to generate embeddings for images, and we evaluate VolcanoML with and without the enrichment of “smote_balancer” operator.

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 69.7% obtained by auto-sklearn without considering embeddings.

5.4 Comparison with 4 Industrial Platforms

In addition, we run additional experiments on six Kaggle datasets to compare VolcanoML with four commercial AutoML platforms: 1) Google Cloud AutoML, 2) Microsoft Azure Automated ML, 3) Oracle data science, and 4) Amazon AWS Sagemaker AutoPilot. Here, we anonymously refer to these platforms as Platform 1-4. Figure 6 show the results, and the Appendix contains the experiment details. We observe that, given the same time budget (i.e., fix the x-axis to some time budget), VolcanoML is at least comparable with, often outperforms, the considered commercial platforms.

6 CONCLUSION

In this paper, we have presented VolcanoML, a scalable and extensible framework that allows users 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 decomposition strategies that also lead to performance-wise better ML pipelines, compared to state-of-the-art AutoML systems.
