CAPS: Cost-Aware ML Pipeline Selection

Antonios Kontaxakis
antonios.kontaxakis@ulb.be
Université Libre de Bruxelles
Brussels, Belgium
Universitat Politècnica de Catalunya
Barcelona, Spain

Dimitris Sacharidis
dimitris.sacharidis@ulb.be
Université Libre de Bruxelles
Brussels, Belgium

Alberto Abelló
alberto.abello@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

Sergi Nadal
sergi.nadal@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

Alkis Simitsis
alkis@athenarc.gr
Athena Research Center
Athens, Greece

ABSTRACT

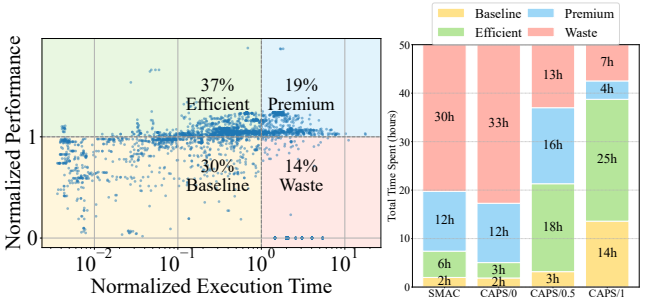
We present CAPS, a novel system for cost-aware pipeline selection in automated machine learning (AutoML). CAPS' approach is orthogonal to the search strategies used by existing AutoML frameworks, enabling seamless integration with them. This integration benefits AutoML systems by reducing computational waste—time spent evaluating inefficient pipelines or exceeding time or memory constraints—through lightweight performance and cost estimation. To incorporate cost-awareness into AutoML, CAPS represents a set of pipelines as a directed hypergraph, estimates execution times for each function, and formulates the optimal subset selection as a constrained prize-collecting optimization problem. CAPS employs a simple yet effective greedy algorithm to approximate this optimal subset. Our evaluation shows that when CAPS prioritizes cost, it reduces waste up to 4×—saving up to 23 hours in 50 hours of pipeline evaluation. When CAPS balances the trade-off between performance and cost, it consistently achieves performance improvements of 5% to 15% over the state of the art within the same time budget across datasets. It also demonstrates that CAPS is compatible with complementary efficiency techniques such as early stopping and warm-starting, achieving speedups of up to 4.7×.

PVLDB Reference Format:
Antonios Kontaxakis, Dimitris Sacharidis, Alberto Abelló, Sergi Nadal, and Alkis Simitsis. CAPS: Cost-Aware ML Pipeline Selection . PVLDB, 19(7): 1427 - 1440, 2026. doi:10.14778/3801059.3801060

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/akontaxakis/CAPS.

1 INTRODUCTION

Selecting an appropriate machine learning (ML) pipeline involves searching through a large number of alternatives that vary in the choice and parameterization (e.g., hyperparameter tuning, learner selection) and data preparation steps (e.g., preprocessing, feature engineering) [32]. The complexity and increasing demand for this task has driven the development of automated ML (AutoML), a research area focused on automating ML pipeline design by systematically exploring the space of possible configurations with minimal human intervention [14]. Although there is a wide range of AutoML approaches, they can generally be viewed as iterating until either the search budget is exhausted or a suitable pipeline is identified. Each iteration consists of two steps: *generation* (GEN) (a set of candidate pipelines is produced for further investigation); *evaluation* (EVL) (these pipelines are executed). After EVL, metadata on their performance is recorded to inform and guide the next GEN step.



(a) SMAC (b) SMAC vs. CAPS

Figure 1: A categorization into Baseline, Efficient, Premium, and Waste of ~3,000 pipelines explored by SMAC (10 ML tasks, 50-hours budget). (a) Each pipeline is represented as a point based on performance and execution time. (b) Time allotted by SMAC compared to three variants of CAPS.

Motivation. Although AutoML needs minimal human input, it incurs high computational costs in time and money [18, 25, 39]. The number of pipeline evaluations depends on the complexity of the task and the desired outcomes, with popular systems performing hundreds to thousands of evaluations per dataset [8, 26]. It is common for only a small number of pipelines (the *needle*) to be selected for deployment from among the thousands generated (the *haystack*). To assess the scale and impact of computational costs, we experimented with ten ML tasks using the pipeline space described in Section 5. For each, we used the AutoML tool SMAC [23] to explore the space within a 5-hour budget. We recorded performance (classification accuracy) and execution time for every evaluated pipeline. For cross-task comparison, we normalized both by dividing each

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801060

value by the task-specific mean (e.g., a normalized execution time of 2 indicates twice the average for that task).

Overall, we evaluated ~3,000 ML pipelines. In Figure 1a, each is plotted in a performance-execution time plane divided into four quadrants. The *Baseline* quadrant includes pipelines with below-average performance and execution time, typically considered naïve or baseline solutions. The *Efficient* quadrant with below-average cost and above-average performance pipelines, representing cost-effective alternatives. The *Premium* quadrant consists of pipelines with an above-average cost and performance, often delivering best results at a higher cost. The *Waste* quadrant includes pipelines with an above-average cost and below-average performance, highly inefficient, and undesirable options. Our experiment shows ~14% of the pipelines fall in the Waste quadrant[1], yet they consume 60% of the total computational cost (30 of 50 hours allocated). In order to make better use of the wasted resources, AutoML systems should integrate the execution cost of pipelines in the process of generating high-performing candidate pipelines. The few past efforts that employ cost-aware methods [36, 42] present two limitations: (a) they are restricted to search strategies in the GEN step and AutoML tasks (e.g., hyperparameter tuning or learner selection), and (b) they rely on rudimentary cost estimates, e.g., treat the pipeline as a black box and model its cost with a Gaussian process surrogate [36], or use past runs of the same learner to predict the time required to outperform the current best pipeline [42].

Our Approach. We present *Cost-Aware ML Pipeline Selection* (CAPS), a framework that introduces cost-awareness into AutoML systems. The central idea is an intermediate *selection* step (SEL) placed between GEN and EVL. SEL prioritizes the pipelines produced by GEN and selects a subset for evaluation. By decoupling selection from generation, each step can pursue distinct objectives: GEN focuses on generating diverse and promising candidates through exploration and exploitation, while SEL emphasizes a desired performance-cost trade-off in prioritization. For example, by increasing the emphasis on cost (CAPS/1 in Figure 1b), we reduce waste by 23 hours and can thus explore 10× more pipelines.

CAPS relies on estimating both the performance and execution cost of a pipeline. Estimating the execution cost of a pipeline is challenging, and previous methods [36, 42] fall short. Rather than treating the pipeline as a black box and estimating its overall cost, CAPS estimates the execution time of each individual function within the pipeline, and integrates with the execution environment to collect fine-grained measurements from past pipeline runs. This granularity is essential, as modern execution environments apply optimizations such as materialization, reuse, and equivalence [6, 17, 43] across pipelines, which can obscure true per-function costs.

From the GEN step output, CAPS builds a directed hypergraph for generated candidate pipelines, where each hyperedge corresponds to a function within a pipeline, while each vertex represents an artifact (such as a dataset, fitted model, or evaluation metric) produced or consumed by these functions. The hypergraph is annotated with hyperedge *costs*, reflecting the estimated execution time of each function, and vertex *prizes*, representing the performance benefit of reaching a particular artifact. CAPS formulates a constrained optimization problem to select the subgraph balancing performance and cost. Since this prize-collecting problem is NP-hard, CAPS uses a simple yet effective greedy algorithm to approximate the optimal solution. We illustrate this with an example.

Example 1. *Figure 2a illustrates an AutoML system call for a single iteration, generating three pipelines from a space with two steps: preprocessing (apply* `PCA` *or skip) and classification (*`DecisionTree` *or* `RandomForest`*). For clarity, we assume that each operator has a single fixed parameterization.*

Figure 2b lists the function calls for each of the three generated pipelines. Pipeline p_1 applies `PCA` *followed by* `DecisionTree`*, p_2 applies* `PCA` *followed by* `RandomForest`*, and p_3 directly applies* `RandomForest` *without preprocessing. Function calls shared across all three pipelines are shown in white boxes, those shared by two pipelines in green, and pipeline-specific functions in blue. Note that while the blocks t_6–t_7 and t_8–t_9 involve identical function calls, they are treated as distinct due to operating on different input artifacts.*

Figure 2c depicts each pipeline as a directed hypergraph. For example, in pipeline p_1, the function `split(data)` *is represented as a multi-output hyperedge t_1 that produces artifacts* `train` *and* `test`*, corresponding to vertices v_1 and v_2, respectively.*

CAPS builds a unified graph by merging the three pipeline hypergraphs, annotating hyperedges with estimated execution cost. Target vertices (v_7, v_9, and v_{11}) are annotated with estimated performance of the model on the test set, derived from the respective `score` *calls. These annotations, in red for estimated costs and green for estimated performances, are illustrated in the hypergraph in Figure 4a.*

Assuming CAPS must select two of the three generated pipelines for evaluation, each possible selection corresponds to a subhypergraph, as illustrated in Figure 2d. The estimated cost of a subhypergraph is computed as the sum of the costs of its hyperedges divided by the number of selected pipelines, while the estimated performance is given by the sum of the prizes on its target vertices divided by the number of selected pipelines. CAPS selects pipelines p_1 and p_2 when prioritizing cost, and pipelines p_1 and p_3 when prioritizing performance.

In practice, CAPS does not enumerate all possible subhypergraphs. Instead, it uses a greedy algorithm shown in Figure 4a to identify the preferred subset of pipelines efficiently, either p_1, p_2 or p_1, p_3, depending on the specified performance-cost trade-off.

Contributions. We introduce CAPS, a novel framework that enables AutoML systems to select ML pipelines based on a user-specified trade-off between performance and execution cost. We integrate CAPS into two popular AutoML systems with fundamentally different GEN strategies: SMAC [23] (uses Bayesian optimization) and TPOT [26] (uses a genetic algorithm). Moreover, we enhance CAPS with two techniques aimed to further improve efficiency: *warm starting* (using meta-learning as in Auto-sklearn [8]), and *early stoping* (using successive halvening as in Hyperband [21]). Our experimental evaluation across ten dataset-task combinations involves AutoML systems such as SMAC [23], TPOT [26], Hyperband [21], BOHB [7], Auto-sklearn [8], and reveals that:

- CAPS finds a pipeline with the best observed performance up to 5× faster when integrated with SMAC, and up to 2× faster with TPOT. To discover a pipeline with near-optimal performance

[1]The Waste quadrant also includes ML pipelines that failed to complete due to timeouts or excessive memory consumption.

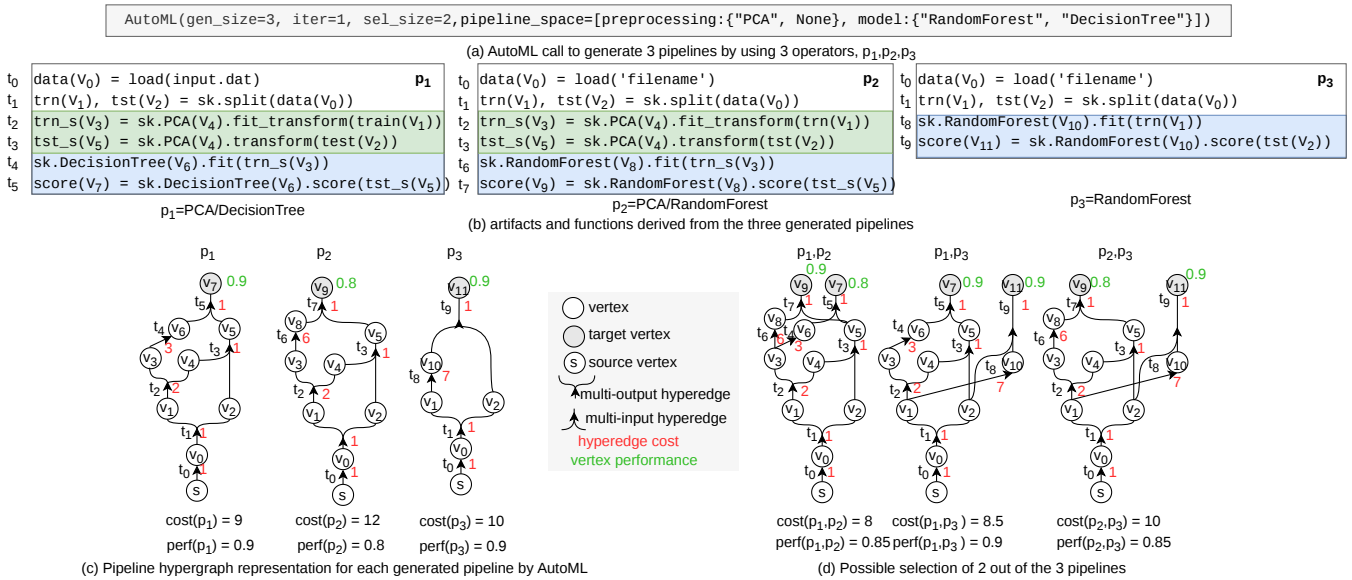


Figure 2: Running example: (a) Statement for a classification problem with one pre-processing step and two classifiers; (b) Three generated pipelines p_1, p_2, p_3; (c) Each pipeline represented as a directed hypergraph, where vertices denote artifacts; (d) Given two out of three pipelines to select, CAPS proposes p_1, p_2 when cost is prioritized, and p_1, p_3 when performance is prioritized.

(i.e., within 96% of the best observed), CAPS achieves up to 4× speed-up with SMAC and up to 10× speed-up with TPOT.
- When prioritizing cost, CAPS enables faster pipeline exploration, achieving up to 10× higher throughput.
- Under a tight time budget (e.g., one hour), CAPS allows the AutoML system to find pipelines with up to 5% better performance, regardless of the GEN strategy used.
- Warm starting (Auto-sklearn [8]) helps CAPS achieve up to 3× speedup. In contrast, for SMAC the speedup is only 1.8×.
- Early stoping (Hyperband [21]) helps CAPS achieve up to 4.7× speedup. In contrast, for SMAC the speedup is only 1.2×.
- CAPS introduces only negligible overhead to an AutoML system, two orders of magnitude smaller than the evaluation step, adding just a few seconds even when selecting hundreds of pipelines.

Outline. Section 2 reviews AutoML systems. Section 3 introduces the problem of cost-awareness in AutoML. Section 4 details the CAPS framework. Section 5 presents a detailed evaluation, and Section 6 concludes the paper.

2 RELATED WORK

AutoML Search Strategies. AutoML systems assist ML developers in the key tasks of data pre-processing, learner selection, and hyperparameter tuning. Yet, few support all of them. At their core, AutoML systems are similar in that they try to find the best ML pipeline inside a search space. Thus, they can be classified based on the search strategy they use.

Random search. Include grid search that exhaustively explores the pipeline space, and random search that considers randomly selected pipelines, such as H2O [20] for hyperparameter tuning and learner selection.

Bayesian optimization. Use a probabilistic surrogate model to predict pipeline performance and guide the search. Such techniques are particularly effective for hyperparameter tuning [5], but some also specialize in pre-processing [12, 15]. Surrogate models, such as tree-structured Parzen estimators and Gaussian processes, struggle with high dimensional search spaces, categorical attributes, and hierarchical structures across all four AutoML tasks. To address this, random forests are widely used as surrogates [13], with SMAC [23] being the most notable implementation, adopted by AutoML systems like AutoSklearn [8], AutoPytorch [46], and AutoWeka [38].

Genetic algorithm. Use mutation and pipeline crossover to evolve better solutions. TPOT [26] is the best-known AutoML system in this class, using scikit-learn pipelines [35] and capable of addressing all AutoML tasks. It represents ML pipelines as trees, nodes correspond to pipeline steps and their configuration, and edges represent how steps are connected. Mutation may change a step's configuration, replace it with a compatible operation, or insert or delete steps. Crossover swaps subtrees between two pipelines. GAMA [11] has a similar scope to TPOT but uses a different genetic algorithm.

Bandit. Allocate more resources to promising solutions, and are predominantly used for learner selection and hyperparameter tuning. In this context, each arm of the bandit corresponds to a learner, and pulling an arm means executing the learner with a different configuration. Successive halving is the key idea and is used in several systems, such as Auto-sklearn, AutoKeras, and Ray Tune [22]: after executing a number of learners, a proportion (typically half) of the worst performing learners are discarded and will not be considered in the future. Hyperband [21] is an adaptive extension of this idea, invoking multiple rounds of successive halving.

Hybrid. They select among different search strategies, such as BOHB [7] that combines Bayesian optimization with Hyperband.

Efficient AutoML. Regardless of the search strategy, AutoML systems are hurt from the computational cost of training and evaluating thousands of models. Past research showed that a single Random Forest model could outperform them in many cases [10]. Several directions are adopted to improve AutoML efficiency.

One direction is to reduce the pipeline execution cost by restricting the data size or the training duration: *Sampling* techniques reduce the size of training data [3, 19, 27, 47], and hence the execution cost of the pipelines (e.g., FLAML [42] and Alpine [33]); *Early stopping* refers to termination of non-promising pipelines, and is compatible with all search strategies (e.g., GAMA [11], Hyperband [21], and BOHB [7]); *Epoch reduction* is a related idea that performs training for few epochs (e.g., [42]).

Another strategy is *meta-learning*, which considers knowledge from past AutoML tasks into the search strategy [31, 44, 45], and can be used on its own (i.e., no informed search) or integrated in systems such as AutoSklearn [8], AutoPytorch [46], or Alpine [33] to improve initial candidate selection. Training a meta-learner requires extensive datasets and experiments, making it less adaptable to new objectives or search spaces. In practice, it is limited to pipelines with enough historical data (e.g., when AutoSklearn [8] is given a custom search space, its meta-learning component is disabled).

Other more general ideas that also apply to AutoML systems include *materialization* and *reuse* of intermediate results [6, 28], operator *equivalence* exploitation [1, 17, 37], and execution *reordering* [24]. Lima [28] tracks lineage within a single pipeline, but does not allow reuse across pipelines. MISTIQUE [41], Helix [43], and Collab [6] focus on reuse of artifacts across pipelines, with Helix using polynomial-time algorithms to optimize it. Raven [1] and Keystone [37] explore operator equivalences, while HYPPO [17] integrates equivalence exploitation with hypergraph representations.

Cost-aware AutoML. Prioritize the execution of pipelines that are promising and cheap. For hyperparameter tuning with Bayesian optimization, EIperSec [36] introduces an acquisition function that normalizes expected improvement by estimated execution cost, using a separate model for cost prediction. FLAML [42] extends this idea to learner selection through the Estimated Cost for Improvement (ECI), which estimates the cost required for each learner to surpass the current best, based solely on past executions.

Research value. CAPS advances cost-aware AutoML in several ways. It addresses the general pipeline selection problem rather than a specific pipeline stage. It is orthogonal to the search strategy and thus complements related approaches. Unlike prior work, CAPS achieves more accurate cost estimation by combining estimators at the *function level* instead of the *pipeline level*. This is key, as function-level estimators can leverage past AutoML runs and generalize to unseen data—a novel form of meta-learning.

3 PROBLEM FORMULATION

This section describes the pipeline configuration space and the AutoML problem, where current systems follow a two-step cycle of generation (GEN) and evaluation (EVL). Our work introduces an additional step, selection (SEL), placed between GEN and EVL.

3.1 Pipelines and Pipeline Search Space

ML pipelines are generally expressed either *implicitly* or *explicitly* [29]. Implicit is the predominant approach in AutoML systems, with *declarative pipelines* describing the pipeline as a sequence of *operators*, which can perform data preprocessing (e.g., `StandardScaler`), feature engineering (e.g., `PCA`), or model fitting (e.g., `RandomForestClassifier`). Each operator has *configuration* options (e.g., number of estimators in `RandomForestClassifier`).

Without loss of generality, here, we adopt the terminology of `scikit-learn.pipelines` for declarative pipelines, where each operator implements several functions, such as `fit`, `transform`, `predict`, and `score`. Also, a declarative pipeline is associated with pipeline-level actions, such as `fit`, `transform`, and `predict`.

Example 2. *A declarative pipeline for a binary classification ML model using a decision tree is expressed as* `[PCA, DecisionTree(DT)]`. *Suppose we want to train the model and compute the accuracy score, and thus invoke the* `score` *action. This action is translated into a sequence of function calls:* `PCA.fit`, `PCA.transform`, `DT.fit`, *and* `DT.score` *as shown in pipeline p_1 in Figure 2b.*

An action on a declarative pipeline along with a configuration of its operators corresponds to an explicit pipeline, which we hereafter simply refer to as a *pipeline*. AutoML systems explore a *pipeline space PS*, which contains a set of pipelines that can be instantiated for the problem at hand. *PS* is defined by specifying the operators that a pipeline can include (e.g., `PCA`, `XGBoost`); their configuration space, such as hyperparameters and their domain (e.g., the number of components in `PCA`, maximum depth in `XGBoost`); and any necessary constraints, such as ordering (e.g., data preprocessing operators must go before feature generation operators).

3.2 The Pipeline Selection Problem

Finding an efficient pipeline in *PS* is often cast as a black-box optimization problem [23]. Modern AutoML systems tackle this with iterative search strategies (Bayesian optimization, genetic algorithms, bandits) that alternate between *generation* (GEN) and *evaluation* (EVL), progressively focusing on promising regions of *PS*.

GEN performs a search on *PS* and suggests new pipelines to investigate. This step considers the history of past executions to select the next best pipelines to test, often with an exploit-explore tradeoff. With an empty history, it is common that meta-learning techniques are used if possible. Otherwise, a random selection is made. In each iteration, we refer to the generated set of pipelines as P_{GEN}. EVL assesses the model performance of each pipeline in P_{GEN} on the target dataset (e.g., via classification accuracy). We refer to the set of pipelines evaluated as P_{EVL}. AutoML systems also rely on an external parameter N, which determines both the size of P_{EVL} and P_{GEN}; this value remains constant across iterations.

The SEL step. One key limitation of GEN is that it generates pipelines without considering their runtime cost. To address this, we introduce an additional step, SEL, which refines the search by selecting a subset P_{SEL} of the generated pipelines that aim to maximize estimated model performance (*perf*) while minimizing estimated computational cost (*cost*). As formalized in the problem statement below, SEL applies a cost penalty to discourage resource waste, while prioritizing pipelines with high estimated performance to increase the likelihood of finding optimal or near-optimal solutions, thereby accelerating convergence.

Problem (Pipeline selection). *Given the generated pipelines P_{GEN}, our goal is to select a subset $P_{SEL} \subseteq P_{GEN}$ that maximizes the combined objective of performance and cost trade-off, subject to a given number of pipelines:*

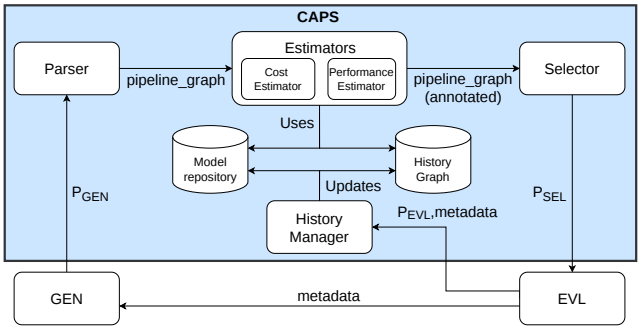


Figure 3: CAPS' system architecture

$$P_{SEL} = \arg \max_{P \subseteq P_{GEN}} ((1-\lambda) \cdot perf(P) - \lambda \cdot cost(P))$$

Subject to the constraint: $|P_{SEL}| = N$

Where $perf(P)$ *represents the average estimated pipeline performance of the subset* P, $cost(P)$ *represents the average estimated pipeline cost of the subset* P, λ *is a weighting parameter that balances the importance of maximizing* $perf(P)$ *and minimizing* $cost(P)$. *The constraint ensures that the selected set contains* N *pipelines, corresponding to the number of pipelines to be evaluated in each iteration of the* `GEN-EVL` *process.*

4 THE CAPS SYSTEM

The system architecture of CAPS is depicted in Figure 3, and can be integrated with any AutoML system adhering to the GEN-EVL cycle. It consists of five components. The *Parser* takes as input a set of pipelines P_{GEN} and constructs their directed hypergraph representation. The *Cost Estimator* is used to annotate this graph with estimated costs at its hyperedges. For this purpose, it uses information from past executions of pipelines stored as *History Graph* and from a *Model Repository* used to predict the cost of unseen hyperedges. The *Performance Estimator* predicts the performance of the pipelines based on the search strategy used by the AutoML system. The *History Manager* maintains the history of executed pipelines. Finally, the *Selector* solves the problem of selecting pipelines based on their predicted performance and cost, subject to a constraint.

4.1 Parser

Following recent work on ML pipeline optimization [6, 17, 43], CAPS adopts HYPPO's hypergraph representation of pipelines [17]. The parser relies on the hosting AutoML framework's *intermediate representation* (its pipeline object) and accepts pipelines in *structured form* (e.g., scikit-learn objects used by TPOT) or as points in a configuration space (e.g., SpaceConfiguration objects used by SMAC). The parser canonically encodes each step as an `(id, opt, type, params, inputs)` tuple (with an id, an operator, a function type, its hyperparameters, and a list of input vertex IDs), and we emit one hyperedge per task. This involves representing pipelines as directed hypergraphs, the functions in a pipeline being hyperedges, and the inputs and outputs of these functions (the artifacts) being vertices. Artifacts can be data, models, or scalars (e.g., single values such as evaluation metrics). We assume that each pipeline terminates after computing a target artifact that represents an evaluation metric assessing the ML model's performance. This hypergraph representation can capture a single or multiple pipelines.

Definition (Pipeline Hypergraph). *A pipeline hypergraph* $P = (V, H)$ *is a directed acyclic hypergraph (DAH), where the vertices* V *correspond to artifacts and the directed hyperedges* H *represent functions that produce or consume these artifacts. A* hyperedge *is a tuple* $h = (id, tail, head, opt, type, params, cost) \in H$ *that connects a set of vertices* $h.tail \subseteq V$, *to a set of vertices* $h.head \subseteq V$. *Each hyperedge is also composed of an id, an operator, a function type, its hyperparameters, and a cost estimation. Each vertex* $v = (id, type, shape)$ *has just an id, a type, and a shape. To simplify explanations and processing, every pipeline hypergraph includes a source phantom vertex identified as* v_s *having no incoming hyperedges, and having outgoing hyperedges to all raw data artifacts. A hypergraph contains a set of target vertices* $V_T \subseteq V$, *where each target vertex corresponds to the evaluation metric of a pipeline. As a result, the number of pipelines represented in a hypergraph equals the number of target vertices* $|V_T|$.

The parser performs two tasks over a set of pipelines: (a) converts them into a *pipeline hypergraph*, and (b) assigns identifiers to its vertices and hyperedges. In the following, we abuse notation and refer to both the set of pipelines and their hypergraph representation using the same symbol P.

The first task involves tracking, based on the action (`fit`, `transform`, or `score`), all functions called within each pipeline. This is straightforward to implement, as it follows the same instructions as the action itself but records the function calls instead of executing them. Since every function has a type (`fit`, `transform`, or `score`), their respective inputs and outputs inherit these types. For example, a `fit` function outputs a fitted operator or model, a `transform` function outputs a transform dataset, and a scoring function outputs a scalar. In Figure 2b, we show an example of the functions $(t_1, \ldots, t_9)$ and their respected input and output artifacts $(v_1, \ldots, v_{11})$ identified by the parser for the pipelines p_1, p_2, p_3.

The parser also assigns ids to the vertices and hyperedges. Hyperedge ids are used to help define vertex ids. Thus, a hyperedge $h \in H$ is identified by concatenating the operator's name, the function type, and the relevant hyperparameters. For example, given the function call `StandardScaler(with_mean=True).fit`, the label of h is `StandardScaler.with_mean=True.fit`.

A vertex $v \in V$, representing an artifact, is assigned a unique id that encodes the sequence of its ancestor edges back to the source vertex v_s (omitted for brevity). For example, if an artifact results from a sequence of edges h_1, h_2, its id would be $h_1.id\_h_2.id$. In practice, all ids are converted to fixed-size hashes.

4.2 History Manager

The History Manager maintains the *History Graph* of all previously executed pipelines and provides information to the *Cost Estimator*. The *History Graph* represents a collection of pipelines and is modeled as a hypergraph H, which combines the functions, artifacts, and dependencies of each evaluated pipeline. As new pipelines are executed, their graphs are incrementally integrated into the history by merging vertices and edges with identical identifiers and adding new elements as needed. The history is updated after each completion of an EVAL step. A simplified version of this exists in most

AutoML systems to track evaluated pipelines. CAPS enhances it by updating the model repository and the *History Graph*.

4.3 Cost Estimator

The Cost Estimator considers the computational cost of individual functions, not the pipeline as a whole. Given a hypergraph P, each hyperedge is annotated with an estimated *cost* for executing the corresponding function, and each vertex with an estimated *shape* –the size per dimension of the artifact (e.g., #rows, #columns).

This annotation process is described in Algorithm 1. The estimator processes the hypergraph in a breadth-first search (BFS) manner adapted to directed hypergraphs, ensuring that artifacts (i.e., vertices) are only visited once all their dependencies have been visited. This is essential because estimating the cost of a function requires first estimating the shapes of all its inputs.

The traversal uses a queue to store visited vertices and proceeds until the queue is empty. Initially, the queue contains only the source vertex v_s, which is marked as visited (lines 1–2). In each iteration, the estimator takes a visited vertex from the queue (line 4) and examines its outgoing hyperedges. For each such hyperedge h (line 5), it first checks that all vertices in the tail of h have been visited, meaning their shapes have been computed (line 6). If so, it invokes `estimate_cost` to estimate the cost of the corresponding function (line 7), then calls `estimate_shape` to estimate the shapes of all the output artifacts (the vertices in head(h)), marks these vertices as visited, and inserts them into the queue (lines 8–11). We next describe the procedures `estimate_cost` and `estimate_shape`.

Estimate Cost. The `estimate_cost` procedure inputs the operator, function type, and associated parameters that correspond to the hyperedge h, and the estimated shape of the artifact corresponding to the input vertices of h. It first checks if the same function with the same input and parameters has been executed before. Thus, it checks if the same edge exists in the history graph by performing a look-up in a hash map (achieved in *amortized* $O(1)$), and if this is the case, it simply retrieves the cost of a past execution. Otherwise, it uses a model from the *model repository* to estimate the cost.

The model repository contains regression models for each operator and function combination. Specifically, the number of models equals the number of operators multiplied by the number of functions per operator. For example, in `sklearn`, the `StandardScaler` operator implements both the `fit` and `transform` functions. Consequently, two models are stored in the repository: `StandardScaler_fit` and `StandardScaler_transform`. Each model uses the operator's configuration parameters as features, along with two data-related features: the number of samples and the number of features in the input. Lastly, we collect training data for each regression model after each Eval step. Once more than ten new training data points are collected, the relative error in the new training data is computed, and retraining is triggered if it is above 20%.

Estimate Shape. The estimate_shape procedure inputs the operator, function type, its associated parameters for the hyperedge h, and the shape of the input vertices. Its implementation differs based on function types (i.e., fit, predict, transform, evaluate). Fit functions always output a fitted operator. Predict functions produce one-dimensional data with a shape of (#input.samples). Transform functions are more complex, as their output often matches the input shape (#input.samples, #input.features). However, exceptions exist, such as Principal Component Analysis (PCA), which outputs a shape determined by configuration parameter #PCA.components, resulting in (#input.samples, #PCA.components). Our current implementation allows users to provide a custom estimate_shape procedure to handle such variations.

Algorithm 1: Estimate Cost of Pipeline Graph

```
   Input  : P: Pipeline_graph, MR: Model_repository, H: History_graph
   Output : P: Pipeline_graph_annotated
1  visited = [P.get_source_vertex()]
2  Q.enqueue(P.get_source_vertex())
3  while Q ≠ ∅ do
4      v = Q.dequeue_one()                          // Pick a visited vertex from Q
5      foreach h ∈ v.outgoingHEdges() do
6          if ∀u ∈ h.tail : u ∈ visited then
7              h.cost = estimate_cost(h.opt, h.type, h.params, h.tail.shape,
                 MR, H)
8              foreach v' ∈ head(h) do
9                  v'.shape = estimate_shape(h.opt, h.type, h.params,
                     h.tail.shape, H)
10                 visited.append(v')
11                 Q.enqueue(v')
12 return P
```

4.4 Performance Estimator

This component provides the estimated $perf(p)$ for the model performance of a given pipeline p. CAPS does not directly make the estimation, but rather relies on the estimations obtained from the GEN step of the AutoML cycle. We make two distinctions.

If the search strategy of the GEN step is based on Bayesian optimization (e.g., as in SMAC), the estimated performance is taken directly from the surrogate model of the Bayesian optimization, whose goal is exactly that: to estimate the performance of a pipeline.

If GEN uses genetic search (e.g., TPOT), there is no estimation of the performance of a pipeline, and instead, the actual performance after the EVL is used as its fitness. Since a pipeline generated in GEN is the result of mutating a single parent pipeline, or crossing over multiple parent pipelines [26], a generated pipeline should improve upon its parent(s). Hence, we estimate the performance of a pipeline p as the maximum actual performance among its parents.

Although CAPS could easily integrate other estimation techniques, our evaluation verifies that our approach is practical and efficient, distinguishing potentially good pipelines from bad ones without having to predict their actual performance.

4.5 Selector

Given a set of generated pipelines P, the goal of the selector is to identify a subset of them $P' \subseteq P$ for execution. The idea is to select pipelines on the basis of both their estimated performance and cost. We formulate this task as a graph optimization problem.

Problem Formulation. The selector operates on the hypergraph representation $P = (V, H)$ of pipelines with target vertices $V_T \subseteq V$. We assume that the hyperedges of P are already annotated by the cost estimator (i.e., each hyperedge $h \in H$ is accompanied with an estimated $cost(h)$). Furthermore, the selector annotates every vertex $v \in V$ with a performance estimation $perf(v)$ as follows: $\forall v \in V_T$ corresponding to the evaluation metric of pipeline p, $perf(v) = perf(p)$; and $\forall v \notin V_T, perf(v) = 0$.

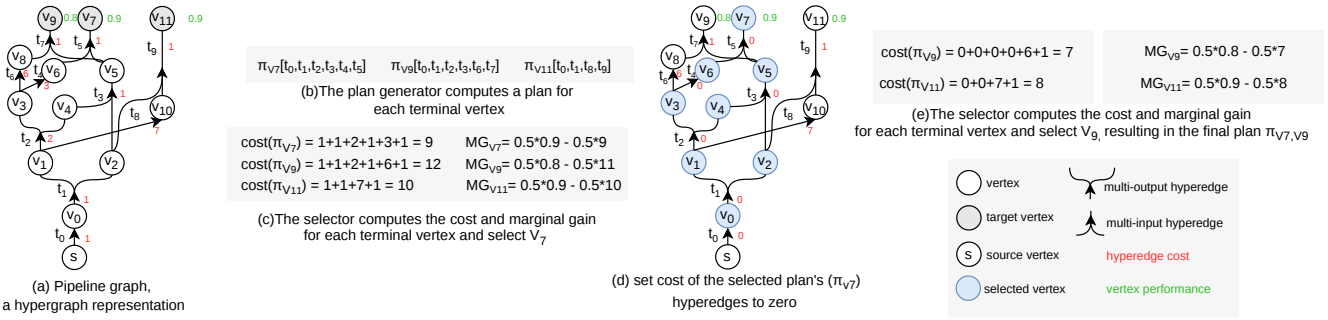


Figure 4: Running example showcasing the select algorithm of CAPS. Given the annotated hypergraph(a), the plan generator computes the plan for each target vertex V_7, V_9, V_{11} (b). The selector computes the marginal gain for each target vertex and selects V_7 (c). It sets the cost of the selected plan's (π_{v_7}) hyperedges to zero (d). Lastly, the selector recomputes the marginal gain for the remaining target vertices and selects V_9, resulting in the final plan π_{v_7,v_9}

When representing functions and artifacts dependencies as a DAH, a critical concept is B-connection, which is analogous to connectivity in regular graphs and is defined recursively [9].

Definition (B-connection). *Given a hypergraph $P = (V, E)$, a target vertex t is* B-connected *to a source vertex s (*$\texttt{B-connected}(s, t)$*) if $t \equiv s$, or $\exists h \in H$, $t \in \texttt{head}(h)$ and $\bigwedge_{v \in \texttt{tail}(h)} \texttt{B-connected}(s, v)$. In a hypergraph, $\forall v \in V : \texttt{B-connected}(v_s, v)$.*

The following definition is the equivalent of a subgraph in directed graphs that encodes a set of paths connecting the source vertex v_s to each target vertex v_t.

Definition (Plan). *Given a hypergraph $P = (V, H)$, we define a* plan *of a subset of target vertexes $V'_T \subseteq V_T$ as a subhypergraph of P, $\pi_{V'_T}(P) = (V', H')$ where the following holds:*

- $V' \subseteq V \wedge H' \subseteq H$; *and*
- $V'_T \cup \{v_s\} \subseteq V'$; *and*
- $\forall v \in V' : \texttt{B-connected}(v_s, v)$ *in* $\pi_{V'_T}(P)$; *and*
- $\forall v \in V' \setminus \{v_s\} : \exists! h \in H'$ *so that* $v \in head(h)$; *and*
- $\forall v \in V' : (\nexists h \in H' : v \in h.tail \rightarrow v \in V_T)$

A hypergraph constructed from a set of pipelines always forms a plan. Conversely, not every subhypergraph corresponds to a subset of pipelines. For a subhypergraph to represent a valid subset, it must be a plan—in which case, the subset of pipelines it encodes are those whose target vertices remain in the subhypergraph.

Note that the average estimated pipeline cost of plan π is given by the sum of the *cost*-values of its hyperedges divided by the number of target vertices, i.e., $cost(\pi) = \frac{1}{|V'_T|} \sum_{h \in H'} cost(h)$. Further, observe that the average estimated pipeline performance of π is the average of the *perf*-values of its target vertices, i.e., $perf(P') = \frac{1}{|V'_T|} \sum_{v \in V'_T} perf(v)$.

Using hypergraph notation, the pipeline selection problem can be reformulated as follows[2].

Problem. *Given a hypergraph $P = (V, H)$, and a target number N of pipelines to select, the pipeline selection problem is to find a plan $\pi_{V'_T}(P) = (V', H')$ that maximizes the objective:*

$$(1 - \lambda) \sum_{v \in V'_T} perf(v) - \lambda \sum_{h \in H'} cost(h),$$

[2]The common factor $\frac{1}{|V'_T|}$ is omitted from the objective.

Algorithm 2: Pipeline Selection Algorithm

Input: P: *Pipeline_graph* annotated, N: number of pipelines to select, λ: cost weight
Output: π: plan

```
1  Π ← Vertex_Plan_Generator(P)                          // Get plans for each vertex
2  π ← ∅ // Initialize plan
3  V_T ← get_target_artifacts(P)
4  for k ∈ [1..N] do
5      foreach v ∈ V_T do
6          v.MG ← (1 − λ) · perf(v) − λ · cost(Π[v])     // Compute marginal gain
7      v* ← argmax_{v∈V_T} v.MG
8      V_T ← V_T \ {v*}                                  // Remove v* from target artifacts
9      π ← π ∪ Π[v*]
10     set_cost_to_zero(Π, Π[v*]) // Set cost to zero for all other uses of edges in Π[v*]
11 return π
```

Subject to the constraint: $|V'_T| = N$.

We note that this problem is related to the *net worth maximization problem* [16] for regular graphs, which is known to be NP-hard, with the additional constraint that the subgraph must meet the requirements for being a plan.

Algorithm: Pipeline Selection. We use an effective greedy algorithm whose pseudocode is in Algorithm 2. It takes as input a hypergraph representing all possible pipeline configurations, the constraint N, and a parameter λ to balance performance and cost estimations during selection. The algorithm progressively builds and outputs a plan $\pi_{V'_T}$ (or simply π) for the selected V'_T target vertices, representing the selected pipelines. An example of the selection algorithm is shown in Figure 4, as a continuation of the one in Figure 2. In Figure 4a, we show the annotated input hypergraph representation P of the pipelines with targets $V_T = v_7, v_9, v_{11}$.

The Selector first receives the hypergraph P, representing the $|V_T|$ possible pipelines, and uses Algorithm 3 to retrieve a plan for each vertex in P, enabling the estimation of the cost associated with computing this vertex as the sum of the costs of the hyperedges in its plan (line 1). Figure 4b shows an example plan for each target vertex; we list only the hyperedges in each plan (without costs).

After, the output plan is initialized, the target vertices V_T are retrieved from P and a for loop is initiated, which computes the marginal gain (MG) for each target vertex, considering both performance and cost estimations, weighted by λ (lines 5–6). The hyperedges participating in this vertex plan are used to estimate the cost, $cost(\Pi[v]) = \sum_{H \in \Pi[v]} H.cost$ (line 6). In Figure 4c, we compute the marginal gain of each plan, assuming a $\lambda = 0.5$. The target resulting in the highest MG is v_7.

Algorithm 3: Vertex Plan Generator

Input: P: Pipeline_graph
Output: List of plans Π where each $\pi_v \in \Pi$ is a plan $\pi_v = (V_{\pi_v}, H_{\pi_v})$

```
1  v_s ← P.get_source_vertex()
2  π_{v_s} ← {v_s}
3  Π ← {v_s : π_{v_s}}
4  visited = [v_s]
5  Q.enqueue(v_s)
6  while Q ≠ ∅ do
7      v ← Q.dequeue_one()                          // Pick a visited vertex from Q
8      foreach h ∈ v.outgoingHEdges() do
9          if ∀u ∈ h.tail : u ∈ visited then
10             π* ← ∅                                // Temporary plan
11             foreach u ∈ h.tail do
12                 π* ← π* ∪ Π[u]
13             π* ← π* ∪ h.head ∪ {h}
14             foreach v' ∈ h.head do
15                 visited.append(v')
16                 Q.enqueue(v')
17                 Π[v'] = π*
18 return Π
```

After choosing the target vertex that maximizes the marginal gain (line 7), this is removed from the set of target vertices (line 8) and its plan is integrated with π (line 9). Furthermore, the costs of the plans for the remaining vertices is reestimated by identifying the hyperedges that take part in the already selected plan and updating the plans of the remaining vertices by setting the cost of the already selected hyperedges to zero (line 10). This process is repeated iteratively until the N of pipelines are selected.

Algorithm: Vertex Plan generator. Algorithm 3 receives a pipeline hypergraph P and, by performing a single BFS-like traversal (similar to when estimating the cost in Algorithm 1), it computes the plan for each vertex in P. It maintains a list Π of key-value pairs, with the key being a vertex v and the value being its plan π_v. The algorithm starts by initializing the entry for the source vertex v_s, whose plan π_{v_s} is the vertex itself, adds it to list Π, marks v_s as visited and adds v_s into a queue Q (lines 1–5). After a while loop is initiated (line 6), which in each iteration, dequeues one vertex v (line 7), and checks if any outgoing hyperedge h from v (line 8) is ready to be traversed, i.e., all vertices in its `h.tail` were visited (lines 9). If all were visited, meaning their plans were computed and stored in Π, then the plan for each vertex in `h.head` is ready to be computed (actually the same for all of them). Thus, to compute the plan of `h.head`, we integrate the plans of each vertex in `h.tail`, adding h and `h.head` to complete the plan (lines 10–13). Lastly, since the plan for each vertex in `h.head` is computed, these vertices are marked as visited, added to the queue, and their plans added to Π (lines 14–17).

Complexity Analysis. The computational complexity of Algorithm 1, and Algorithm 3, depends on the number of vertices V and hyperedges H in the pipeline representation P. Both algorithms perform a single BFS-like pass over the hypergraph, visiting each hyperedge and vertex once, resulting in a time complexity of $O(|H| \cdot |V|)$.

The pipeline selection algorithm (Algorithm 2) depends on the number of target vertices $|V_T|$ and hyperedges $|H|$ in P and N, the number of pipelines to select. It performs N iterations, and in each iteration, it computes the marginal gain for $|V_T|$ vertices and updates their plans. In the worst case, all hyperedges $|H|$ in P are updated, leading to a worst-case time complexity of $O(N \cdot |V_T| + |H|)$.

Dataset	philippine	jasmine	jannis	helena	fabert	dilbert	robert	albert	digits	dionis
# Rows	5,832	2,984	83,733	65,196	8,237	10	10	425,24	1,797	416,188
# Columns	309	145	55	28	801	2,001	7,201	79	65	61

Table 1: Dataset details used in experiments

5 EXPERIMENTAL EVALUATION

Implementation. CAPS is implemented in Python (3.10), uses NetworkX (3.1) for graph representation, and integrates seamlessly with AutoML frameworks: TPOT (0.12.2) [26] for genetic search and SMAC (2.2.0) [23] for Bayesian search. CAPS interfaces with HYPPO [17] extending its Parser and History Manager.

Cluster. Experiments ran on the DRAGON2 cluster [40], which runs Slurm 24.05.2 on 17 nodes, each with 2× Intel Skylake 16-cores Xeon 6142 processors (2.6 GHz), 192GB RAM (15 nodes) and 384GB RAM (2 nodes), 3.3TB RAID-0 disks, interconnected on 10GbE network running Rocky Linux 8.10 (Green Obsidian).

Datasets. We experiment with ten popular datasets [2] commonly used in AutoML benchmarks, research, and challenges. To ensure a balanced evaluation, we select 50% small to medium-sized ($n_{\text{samples}} \times n_{\text{features}} \leq 10,000,000$) datasets (philippine, jasmine, jannis, helena, fabert) and 50% large datasets (dilbert, robert, albert, digits, dionis). This contrasts with many other AutoML evaluations, where only 10–20% of the datasets are medium-sized and 80–90% small. Table 1 shows datasets statistics: number of rows (instances) and columns (features plus target) for each benchmark.

Operators. We adopt TPOT's standard configuration space comprising 32 operators: 14 preprocessors (6 scalers, 5 transformers, 2 decomposition methods, and 1 encoding), 5 feature selectors (3 statistical and 2 model-based selection), and 13 classifiers (3 linear, 4 tree-based, 3 probabilistic, 1 neural, and 2 other approaches). Each operator includes its standard hyperparameter ranges; e.g., tree-based methods use $max\_depth \in [1, 11]$, regularization parameters $C \in [10^{-4}, 25]$, and ensemble methods default to 100 estimators. All experiments use 3 random seeds for reproducibility. The complete configuration can be found on the paper's companion website [4].

Method. First, we investigate CAPS performance impact on two AutoML search algorithms: genetic algorithms and Bayesian optimization. In addition, we evaluate how CAPS relates to two advanced techniques: early stopping with successive halving and warm-starting with meta-learning. Next, we scrutinize the effectiveness of CAPS components: cost estimator, performance estimator, selector, along with computation and memory overheads.

5.1 Comparison with AutoML Systems

5.1.1 Baselines. Comparing AutoML systems is challenging, as each defines its own search space and hence, it is tough to determine whether a performance advantage is due to a better search strategy or a more effective search space [10]. Hence, for fair evaluation, we ensure that all baselines use a shared search space and examine CAPS's impact when integrated with two strategies: *Bayesian optimization* and *genetic algorithms*. Both are evaluated using the same set of operators and hyperparameters, generating pipelines that include preprocessing, feature selection, and model training.

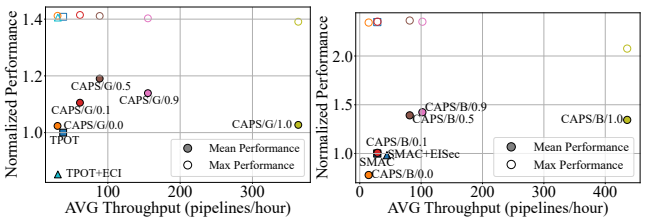


Figure 5: Performance vs. cost: TPOT (left) and SMAC (right)

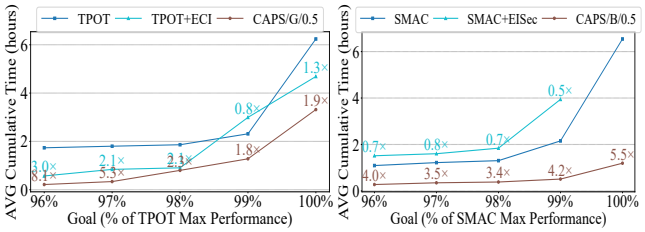


Figure 6: Perf constraint: TPOT (left) and SMAC (right)

Bayesian. Most AutoML systems that employ a Bayesian optimization search strategy (e.g., Auto-PyTorch [46], Auto-sklearn [8], and Auto-WEKA [38]) rely on SMAC [23]. We consider two SMAC configurations: the *default*, which only uses an estimate of improvement, and a *cost-aware variant* that incorporates a cost function to compute the Estimated Improvement per Second (EISec) [36]. The latter trains two surrogate models to predict cost and performance, which are also used by the acquisition function. While SMAC includes built-in support for EISec, it does not mandate specific surrogate model types. We adopt Random Forests for both cost and performance prediction, following SMAC's defaults. Both SMAC and SMAC+EISec are configured with 3 key parameters: the number of initial pipelines to evaluate (set to 20, as in Auto-sklearn), the number of pipelines evaluated before retraining the surrogate model (default: 8), and the choice of acquisition function (expected improvement for SMAC and EISec for SMAC+EISec).

Genetic. Genetic algorithms are used in AutoML systems such as TPOT [26] and GAMA [11]. We focus on TPOT due to its popularity (9K+ GitHub stars) and also implement a cost-aware variant, TPOT+ECI, which incorporates the Estimated Cost for Improvement (ECI) strategy from FLAML [42]. ECI tracks both performance and execution cost across learners by computing deltas; since we operate on full pipelines, we group them by learner so that all pipelines with the same learner share an ECI value. Both TPOT and TPOT+ECI use the same configuration: a population size of 50 and 100 generations, resulting in 5,000 pipelines evaluated per run.

Thus, we evaluate two variations of CAPS: CAPS/B, which integrates our approach with SMAC, and CAPS/G, which integrates it with TPOT. CAPS/G employs a parent-based performance estimator, while CAPS/B uses a model-based estimator, as detailed in subsection 4.4. CAPS leverages the parameter λ to control the trade-off between performance and cost.

5.1.2 Comparison. To demonstrate the practicality and benefits of CAPS within an AutoML system, we first analyze the trade-off between performance and cost. We then evaluate CAPS in two use cases, each reflecting a different constraint scenario.

(a) *Exploring the trade-off between performance and cost*: The user aims to execute a fixed number of pipelines (1,000). For each approach, we report the average and maximum performance achieved, along with its throughput in pipelines per hour.

(b) *Target performance constraint*: The user prioritizes achieving a target performance ranging from 96% of the best-performing pipeline in terms of accuracy to 100%, and we evaluate how many resources different approaches use to achieve that.

(c) *Time budget constraint*: The user aims to identify the best-performing pipelines within a fixed time budget, evaluating scenarios with time limits ranging from 30 minutes to 16 hours.

We present results separately for genetic and Bayesian search. For Bayesian search, we compare SMAC and SMAC with EISec (SMAC+EISec), against CAPS integrated with SMAC (CAPS/B). For genetic search, we compare TPOT and TPOT+ECI, against CAPS integrated with TPOT (CAPS/G). In both cases, we use for CAPS four extreme values of λ (0.0, 0.1, 0.9, and 1.0) and a median value (0.5), which we refer to as CAPS/B/λ and CAPS/G/λ, respectively. Note that we apply min-max normalization to cost, ensuring that $\lambda = 0.5$ represents an equal weighting of cost and performance. As AutoML introduces randomness, we ran each experiment three times with different seeds, and report average values.

(a) Exploring the trade-off between performance and cost. We set a pipeline constraint of evaluating 1000 pipelines for each approach to explore the trade-off between performance and cost (throughput). Pipeline constraints are the default in AutoML systems, as defining an efficient time budget or a target performance constraint requires additional knowledge of the problem's complexity. Figure 5 plots average throughput (pipelines/hour) on the x axis and normalized performance (mean and max, normalized by the baseline's mean performance) on the y axis across datasets and seeds.

In genetic search, Figure 5(left), CAPS/G variants with $\lambda = 0.1$ and $\lambda = 0.5$ offer higher mean performance and higher throughput, outperforming TPOT, TPOT+ECI, and CAPS/G/0.0. CAPS/G/1.0 achieves the highest throughput (10× faster) with slightly lower maximum performance compared to TPOT. CAPS/G/0.9 behanves similar to CAPS/G/1.0, with higher throughput and lower maximum performance than TPOT, presenting a performance-cost trade off. TPOT, TPOT-ECI, and CAPS/G/0.0 achieve competitive maximum performance but suffer from lower throughput, and their inability to handle timeouts reduces their overall mean performance.

In Bayesian search, Figure 5(right), CAPS/B/0.5 and CAPS/B/0.9 offer the best balance between performance and throughput. CAPS/B/1.0 achieves the highest throughput (10× faster) but with reduced maximum performance, while SMAC and CAPS/B/0.0 retain higher maximum performance at the cost of significantly lower throughput. SMAC+EISec is slightly more efficient than SMAC.

Overall, the top-performing approaches are CAPS/G/0.1 and CAPS/G/0.5 for genetic search, and CAPS/B/0.5 and CAPS/B/0.9 for Bayesian search. As discussed in Section 5.2(c), when CAPS is integrated with an AutoML system that makes model-based predictions, such as SMAC, it benefits from improved performance estimation, enabling the use of higher λ values without discarding high-performing pipelines. All in all, CAPS/$\lambda = 0.5$ consistently balances performance and throughput across both search strategies, reinforcing the advantage of combining cost and performance in the search process. Thus, in the remainder of the evaluation, to enhance readability, we further evaluate CAPS with $\lambda = 0.5$.

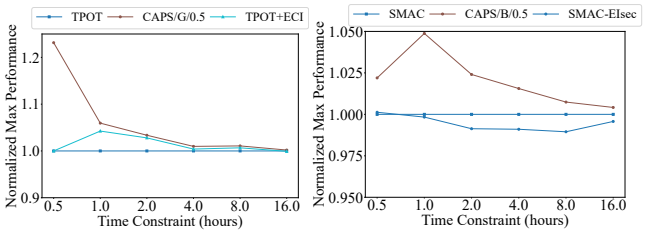


Figure 7: Time budget constraint: TPOT(left) and SMAC(right)

(b) Target performance constraint. An alternative to pipeline constraints is to impose a performance goal and measure the time it takes each method to achieve that goal. Figure 6 shows the average cumulative time (across datasets) required to reach increasing percentages of the maximum performance achieved by the respective baselines after a 24-hour run (TPOT for genetic search, SMAC for Bayesian search). Additionally, we report the average speed-up (across datasets) achieved compared to the respective baseline.

In genetic search, Figure 6(left) shows that CAPS/G/0.5 reaches all target performance constraints (from 96% to 100%) faster than TPOT and TPOT+ECI. The average speed-up of CAPS across the datasets over TPOT ranges from 8.1× at 96% to 1.9× at 100%. While TPOT+ECI performs well up to 98%, its time increases sharply at higher goals. In contrast, CAPS/G/0.5 outperforms the baseline in all constraints, including 100%.

In Bayesian search, Figure 6(right), shows that CAPS/B/0.5 consistently outperforms SMAC and SMAC+EISec across all performance goals. Notably, CAPS/B/0.5 reaches 96% of SMAC's performance up to 4× faster and achieves 100% performance in significantly less time (5.5× faster than SMAC). SMAC+EISec performs worse than SMAC for all performance constraints, as it constantly misses the best-performing executions.

Overall, our finding show that CAPS, with $\lambda = 0.5$ to balance cost and performance, achieves fast convergence. In contrast, approaches that do not balance these factors, such as SMAC+EISec, require significantly more time to reach top performance levels.

(c) Time budget constraint. Finally, we also evaluate each approach under time constraints, a standard criterion in AutoML research. Given a time limit, we take the highest model performance achieved within that constraint and normalize it by the maximum performance achieved by the corresponding baseline (TPOT for genetic search and SMAC for Bayesian search). We report the average normalized maximum performance across datasets and seeds.

In genetic search, Figure 7(left) shows that CAPS/G/0.5 achieves significantly higher normalized performance under the shortest time budget (30 minutes), exceeding 15% improvement across datasets. As the time budget increases, performance differences between methods narrow, and all approaches converge toward the TPOT baseline. This is expected, as all methods explore the same search space and, given time, eventually reach similar performance.

In Bayesian search, Figure 7(right) shows that CAPS/B/0.5 consistently outperforms both SMAC and SMAC+EISec, starting with a 2% performance gain at the 30-minute mark and reaching its peak relative improvement of 5% after one hour of execution. SMAC+EISec shows a slight dip in performance at intermediate budgets (2-4 hours) before recovering at 16 hours. As with genetic search, given time, all approaches gradually converge to similar performance.

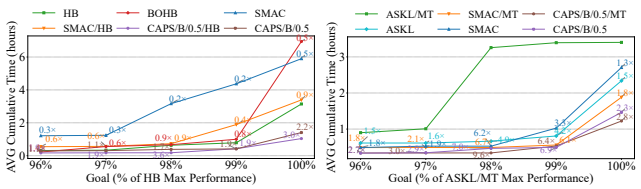


Figure 8: Early stopping (left) and Warm starting (right)

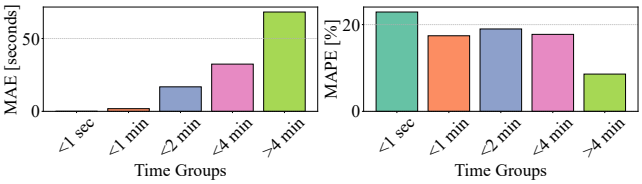


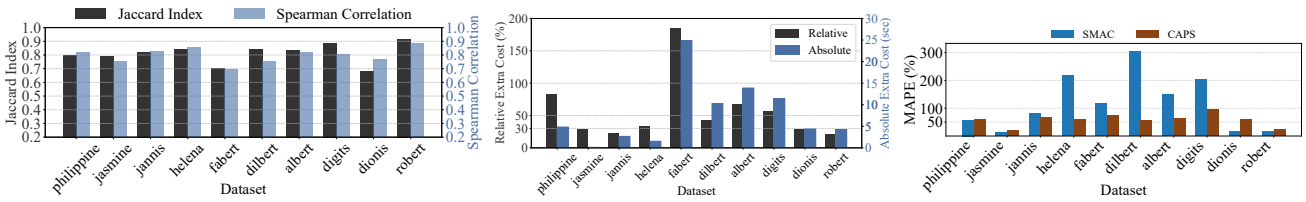
Figure 9: Prediction errors: MAE (left) and MAPE (right)

5.1.3 Advanced AutoML Techniques. As discussed in Section 2, AutoML systems often use performance boosters, such as *early stopping* and *warm starting with meta-learning*. We extend our analysis to evaluate how CAPS is affected by such techniques. Specifically, we consider popular approaches for each: Hyperband [21] and BOHB [7] for the former, and Auto-sklearn [8], for the latter.

Early Stoping. Hyperband [21] (HB) is a bandit-based, early stopping technique that creates buckets and evaluates pipelines with less budget. Here, we consider the dataset size as the budget, so Hyperband trains most pipelines in a subset of the dataset and only the good ones in the full dataset. We also employ an alternative, BOHB [7], that replaces the multi-bandits approach of Hyperband for generating pipelines with a Bayesian model (KDE, kernel density estimator - based surrogate model). For comparison, we consider SMAC with Hyperband (SMAC/HB) using a RandomForest Bayesian approach. We also extend CAPS with Hyperband (CAPS/G/05/HB) using the setup of Section 5.1.2-(target performance constraint), changing the baseline to Hyperband.

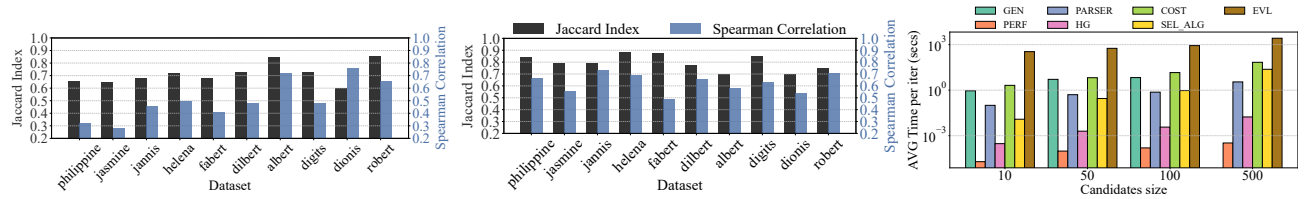
Figure 8(left) shows that all approaches with HB achieve faster results as they are trainning in a subset of data. SMAC does not have any cost awareness and it is almost 10× slower. CAPS without hyperband has benefits over SMAC, but is better than Hyperband after the 99% target, and at 100% has an average speed up of 1.7 across datasets. Lastly, CAPS/B/0.5/HB consistently outperforms Hypeband across targets, starting for 1.7% at 96% and reaching up to 4.7% across speed up across datasets, showcasing the additive betefits of using early-stopping with our selection. Overall, the results show that early stopping provides additional benefits to all methods compared. Still, it is worth noting that CAPS, even without HB, outperforms all approaches both, with or without HB.

Warm Starting. To illustrate how CAPS interacts with warm starting, we use Auto-sklearn (ASKL) [8], which warm-starts via a portfolio-based meta-learning scheme: pipelines are drawn from a predefined set using a k-nearest neighbors (k-NN) model trained on dataset metadata. For our evaluation, we restrict ASKL to operators in our search space (i.e., the portfolio matches the space), disable ensembling, and cap each evaluation at 5 minutes. We consider two ASKL variants: plain ASKL (no meta-learning) and ASKL/MT (with meta-learning). We also integrate the ASKL portfolio into SMAC by



(a) Spearman correlation and Jaccard similarity **(b) Avg relative/absolute error per pipeline** **(c) Cost estimation comparison**

Figure 10: Cost-based measurements



(a) Parent-based performance estimation **(b) Model-based performance estimation** **(c) Average execution time per iteration**

Figure 11: Performance-based measurements and compute overheads

replacing SMAC's initial selection so that pipelines come from the portfolio; as a result, CAPS can select in a cost-aware fashion. We denote meta-learning versions as SMAC/MT and CAPS/MT. We use the setup of Section 5.1.2-(target performance constraint), changing the baseline to ASKL/MT. Each method generates 25 pipelines before the Bayesian search begins (as the default for ASLK).

Figure 8(right) shows that all approaches outperform ASKL/MT. In chasing high-performance pipelines, ASKL/MT suffers the most timeouts (up to 150 pipelines - 12.5 hours), slowing optimization. ASKL and SMAC randomly choose initial pipelines and yield similar gains (1.3× and 1.5× at the 100% target). SMAC/MT randomly selects from a similar portfolio, indicating that random selection can outperform ASKL's k-NN chooser when timeouts are in place. Moreover, CAPS/B/0.5 beats SMAC/MT, showing that cost awareness matters as much as meta-learning under tight timeouts. Finally, CAPS/B/0.5/MT selects from the portfolio in a cost-aware way (by expected cost), reaching 96% 4× faster and 100% 3× faster.

5.2 Scrutinizing CAPS

To further measure the effectiveness of CAPS, we investigate three research questions: (a) How good are the Cost Estimator function-level predictions? (b) How good is the Cost Estimator for ranking and selecting pipelines based on their cost? And how does it compare to other approaches? (c) How good is the Performance Estimator for ranking and selecting pipelines based on their performance? (d) What is the computation and memory overhead introduced by CAPS? (e) How good is our linear heuristic compared to other heuristic approaches, and how does our greedy algorithm compare to a more accurate approach, such as beam search?

(a) Cost Estimator function-level predictions. The Cost Estimator relies on a repository of regression models (one per operator's function) for predictions. To build the repository, we trained popular regressors from scikit-learn with default parameters: LinearRegression, RandomForestRegressor, GradientBoostingRegressor, KNeighborsRegressor, and ExtraTreesRegressor. For each function, we selected the model with the lowest Mean Absolute Percentage Error (MAPE). For evaluation, we used training data comprising 100 points (different operator configurations) per dataset (10 datasets), per operator (32 operators), and per function (1–3 functions), totaling 1000 points per model. The test set includes 10 points per dataset, leaving 910 points for training and 90 for testing. Since execution times range from milliseconds to minutes, to ease the readability, we report MAPE and Mean Absolute Error (MAE) across test set groups: <1 sec, <1 min, <2 min, <4 min, and > 4 min.

Figure 9 shows that the absolute error increases with execution time while the relative error decreases. We aimed for a relative error below 20% at the function level, as execution variance can naturally reach this threshold. The high MAPE for low-cost functions (<1 sec) is negligible, as the absolute error remains under 1 second.

(b) Cost-based pipeline ranking and selection. To assess the Cost Estimator's efficiency at the pipeline level, we evaluate its ability to rank and select top pipelines by estimated cost. Using TPOT, we generate 100 pipelines per iteration for 10 iterations across multiple datasets, then rank and select the 50 cheapest pipelines. We measure ranking performance with Spearman correlation and selection accuracy with Jaccard similarity, averaging these metrics across iterations. We compare our selection with the optimal selection and report the average absolute and relative error per pipeline. To compare with past approaches, we use SMAC that trains a RandomForestRegressor over the whole pipeline space, the same model used in (EISec) [36]. This non-pretrained model is trained after every iteration finishes. For a fair comparison, the model trains on the first 100 pipelines, after which the prediction MAPE is measured.

Figure 10a shows that overall, CAPS achieves a Spearman correlation above 0.7, except for the *fabert* and *dionis* datasets. As Spearman measures ranking accuracy, we further examine Jaccard similarity between the estimated and optimal Top-50 pipelines. Figure 10a shows that overall, the similarity remains above 0.7, and exceeds 0.8 in 7/10 datasets. Figure 10b illustrates the average relative and absolute extra cost per pipeline compared to the optimal selection. In most datasets, our selection performed well with a relative extra cost below 30%. However, in the *fabert* and *albert* datasets

there were relative additional costs of 1× and 2× per pipeline, respectively. This result reflects the expected limitations of CAPS, and highlights a potential improvement opportunity by incorporating dataset-specific features; an interesting topic for future work.

Figure 10c compares the black-box model used by SMAC/EISec, which trains a Random Forest regression model, with our fine-grained approach. Since our model is pretrained, we measure the MAPE of SMAC's predictions after an initial EISec phase of 100 pipelines. We then compute the MAPE over the next 100 pipelines and repeat this process for each of our 10 datasets. CAPS consistently achieves a MAPE below 100% across all datasets, whereas SMAC's model exceeds 100% on *fabert*, *albert*, and *digits*, and even surpasses 200% on *dilbert* and *helena*. This highlights the limitations of relying on a single model to learn such a high-dimensional space.

(c) Performance-based Pipeline ranking and selection. We assess performance-based pipeline ranking and selection using Spearman correlation and Jaccard similarity. Figure 11a shows results using the proposed parent-based estimation for performance. Our heuristic approach achieves a similarity above 0.6 with the optimal selection and exceeds 0.8 for two datasets. However, it fails to achieve an accurate ranking. Notably, within an iteration, multiple pipelines may be generated from the same parent, leading to identical estimated performance values, which rarely hold in practice.

Figure 11b shows the performance estimation when CAPS is integrated with a model-based approach, here, a RandomForest model. Using a predictive model achieves a 0.7 Jaccard similarity across all datasets and surpasses 0.8 in 4 out of 10 cases. While this approach outperforms the heuristic method, it requires constant model retraining at runtime, adding computational complexity. In contrast, parent-based estimation adds no significant complexity or computation overhead, making a practical solution for many use cases.

(d) Computational and Size overhead. Figure 11c, presents the average execution time per iteration for candidate sizes $N = \{10, 50, 100, 500, 1000\}$ on the *jannis* dataset, decomposed into pipeline generation (GEN), evaluation (EVL), and selection (SEL). The SEL step includes performance estimation (PERF), parsing (PARSER), hypergraph checking (HG), cost estimation (COST), and the selection algorithm (SEL_ALG). Evaluation (EVL) dominates runtime, increasing from 68 seconds at $N = 10$ to over 10,617 seconds at $N = 1000$, while GEN (99 seconds at $N = 1000$) and SEL (315 seconds at $N = 1000$) contribute minimally, confirming that the proposed selection introduces negligible overhead compared to EVL.

Within SEL, cost estimation (COST) is the primary contributor, taking 275 seconds at $N = 1000$ due to model retrieval and inference for new hypergraph edges. SEL_ALG scales linearly, growing from 0.1 seconds at $N = 10$ to 30 seconds at $N = 1000$, supporting our complexity analysis. Memory usage for the hypergraph averages 17.739 MB (ranging from 5.112 MB to 101.388 MB for 2101 pipelines), indicating modest space costs relative to computational savings. As EVL remains the main bottleneck, future work could optimize COST through batch model retrieval to further reduce overhead.

(e) Alternative designs for the Selector. In Figure 12, we compare our Greedy algorithm with three alternative design choices, using the setup of Section 5.1.2-(target performance constraint). *TopKPerf* ranks pipelines solely by estimated performance; since cost is ignored, it is neither particularly fast nor effective at achieving high performance. *Perf/Cost* ranks by the ratio of performance to cost, which makes it faster but also biases the search toward the cheapest pipelines, as cost dominates the denominator. Finally, *Beam search* explores a richer set of candidates and consistently finds better solutions than the Greedy baseline, though this comes at the expense of substantially higher computational effort, up to 5× more than Greedy. Note that Beam search with k parameter set to 1 resembles CAPS Greedy. Here, we show Beam search performance with k set to 5; higher k leads to better solutions but at a higher compute cost.

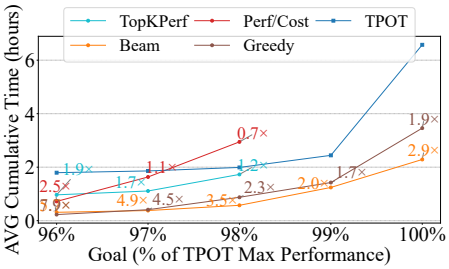


Figure 12: Alternative design choices

Discussion. In CAPS, which adds the SEL step, N remains fixed. GEN generates more than N pipelines, after which SEL selects N. We generate $2N$ candidates, a ratio that performed well empirically. The parameter λ controls the cost–performance trade-off: $\lambda = 1.0$ favors low-cost pipelines, enabling faster discovery of reasonable solutions but potentially missing high-performing ones (Figure 5). Lower values prioritize performance at the cost of longer evaluations.

6 CONCLUSION

We introduced CAPS, a cost-aware AutoML approach optimizing both pipeline performance and computational cost. By integrating an intermediate selection step into existing search strategies (e.g., genetic, Bayesian), CAPS uses a hypergraph to avoid redundant evaluations and a function-level cost estimator to greedily select efficient pipelines. Experiments show that CAPS accelerates genetic and Bayesian searches by up to an order of magnitude and 5x, respectively. It consistently improves performance under tight time budgets and reduces execution time by up to 10x when prioritizing cost. CAPS also supports complementary techniques like warm-starting and early termination. Future work includes incorporating a diversity metric [30, 34] as a third selection objective, extending CAPS's efficiency benefits to ensemble-based AutoML tools.

ACKNOWLEDGMENTS

This work has been partially supported by the H2020-MSCA-ITN-2020 DEDS project (GA 955895), the EU Horizon Europe research and innovation programmes DataGEMS (GA 101188416), WiseFood (GA 101181895), and CREXDATA (GA 101092749), MLEvol (PID2024-156019OB-I00/AEI/10.13039/501100011033), TALC (PID2023-152841/OA-I00/AEI/10.13039/501100011033) and by ERDF, EU. The computation infrastructure has been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region.

REFERENCES

[1] Sadeem Alsudais, Avinash Kumar, and Chen Li. 2023. Raven: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (Seattle, WA, USA) *(HILDA '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597465.3605219

[2] AutoML Challenge. 2025. Datasets. Available at: https://automl.chalearn.org/data.

[3] Maximilian Böther, Ties Robroek, Viktor Gsteiger, Robin Holzinger, Xianzhe Ma, Pınar Tözün, and Ana Klimovic. 2025. Modyn: Data-Centric Machine Learning Pipeline Orchestration. In *Proceedings of the ACM on Management of Data (SIGMOD)*, Vol. 3. https://doi.org/10.1145/3709705

[4] CAPS. 2025. Code repository. Available at: https://github.com/akontaxakis/CAPS.

[5] Anjir Ahmed Chowdhury, Argho Das, Khadija Kubra Shahjalal Hoque, and Debajyoti Karmaker. 2022. A Comparative Study of Hyperparameter Optimization Techniques for Deep Learning. In *International Joint Conference on Advances in Computational Intelligence (IJCACI 2021)*. Springer, 439–450. https://doi.org/10.1007/978-981-19-0332-8_38

[6] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 1701–1716.

[7] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 1437–1446. http://proceedings.mlr.press/v80/falkner18a.html

[8] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2020. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. *arXiv preprint arXiv:2007.04074* (2020).

[9] Giorgio Gallo, Giustino Longo, and Stefano Pallottino. 1993. Directed Hypergraphs and Applications. *Discret. Appl. Math.* 42, 2 (1993), 177–201. https://doi.org/10.1016/0166-218X(93)90045-P

[10] Pieter Gijsbers, Erin LeDell, Jan Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. 2019. An open source AutoML benchmark. *arXiv preprint arXiv:1907.00909* (2019). https://arxiv.org/abs/1907.00909

[11] Pieter Gijsbers and Joaquin Vanschoren. 2021. GAMA: A General Automated Machine Learning Assistant. In *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track*, Yuxiao Dong, Georgiana Ifrim, Dunja Mladenić, Craig Saunders, and Sofie Van Hoecke (Eds.). Springer International Publishing, Cham, 560–564.

[12] Joseph Giovanelli, Besim Bilalli, and Alberto Abelló. 2022. Data pre-processing pipeline generation for AutoETL. *Information Systems* 108 (2022), 101957. https://www.sciencedirect.com/science/article/pii/S0306437921001514

[13] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization (LION)*. 507–523.

[14] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2019. *Automated Machine Learning - Methods, Systems, Challenges*. Springer.

[15] Ihab F. Ilyas, Theodoros Rekatsinas, Vishnu Konda, Jeffrey Pound, Xiaoguang Qi, and Mohamed Soliman. 2022. Saga: A Platform for Continuous Construction and Serving of Knowledge At Scale. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, 2502–2515. https://doi.org/10.1145/3514221.3526049

[16] David S. Johnson, Maria Minkoff, and Steven Phillips. 2000. The prize collecting Steiner tree problem: theory and practice. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, USA) *(SODA '00)*. Society for Industrial and Applied Mathematics, USA, 760–769.

[17] Antonios I. Kontaxakis, Dimitris Sacharidis, Alkis Simitsis, Alberto Abelló, and Sergi Nadal. 2024. HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning. In *Proceedings of the 40th IEEE International Conference on Data Engineering (ICDE 2024)*. IEEE, 221–234.

[18] Andreas Krause and Daniel Golovin. 2014. Submodular Function Maximization. In *Tractability: Practical Approaches to Hard Problems*, Lucas Bordeaux, Youssef Hamadi, and Pushmeet Kohli (Eds.). Cambridge University Press, 71–104. https://doi.org/10.1017/CBO9781139177801.004

[19] Teddy Lazebnik, Amit Somech, and Abraham Itzhak Weinberg. 2022. SubStrat: A Subset-Based Optimization Strategy for Faster AutoML. *Proceedings of the VLDB Endowment* 16, 4 (2022), 772–780. https://doi.org/10.14778/3574245.3574261

[20] Erin LeDell and Sebastien Poirier. 2020. H2O AutoML: Scalable Automatic Machine Learning. *7th ICML Workshop on Automated Machine Learning (AutoML)* (July 2020).

[21] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research* 18, 1 (2018), 6765–6816. http://jmlr.org/papers/v18/16-558.html

[22] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118* (2018).

[23] Marius Lindauer, Katharina Eggensperger, André Biedenkapp, Marius-Leon Duschatzky, Steven Adriaensen, José Jiménez, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. In *Journal of Machine Learning Research*, Vol. 23. 1–9.

[24] Ziniu Liu, Young-Kyoon Park, Juan Camacho-Rodríguez, Mariam Niazi, Woon-Hak Kim, Hyoungshick Kim, and Bongki Moon. 2023. Optimizing Data Pipelines for Machine Learning in Feature Stores. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4230–4242. https://dl.acm.org/doi/10.14778/3625054.3625060

[25] Felix Neutatz, Marius Lindauer, and Ziawasch Abedjan. 2025. How Green is AutoML for Tabular Data?. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*. 350–363. https://openproceedings.org/2025/conf/edbt/paper-97.pdf

[26] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (2016), 485–492. https://doi.org/10.1145/2908812.2908918

[27] David Pérez, Salvador García, and Francisco Herrera. 2022. Auto-CVE: An AutoML Framework with Dynamic Sampling Holdout. *SN Computer Science* 3, 5 (2022), 1–14. https://doi.org/10.1007/s42979-022-01406-4

[28] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-Grained Lineage Tracing and Reuse in Machine Learning Systems *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1426–1439. https://doi.org/10.1145/3448016.3452788

[29] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. 51, 2 (jul 2022), 30–37. https://doi.org/10.1145/3552490.3552496

[30] Lennart Oswald Purucker, Lennart Schneider, Marie Anastacio, Joeran Beel, Bernd Bischl, and Holger H. Hoos. 2023. Q(D)O-ES: Population-based Quality (Diversity) Optimisation for Post Hoc Ensemble Selection in AutoML. In *Proceedings of the Second International Conference on Automated Machine Learning (AutoML)*. 10/1–10/34. https://proceedings.mlr.press/v224/ Available at https://arxiv.org/abs/2307.08364.

[31] Sergey Redyuk, Zoi Kaoudi, Sebastian Schelter, and Volker Markl. 2024. Assisted design of data science pipelines. *The VLDB Journal* 33, 5 (2024), 1129–1153. https://doi.org/10.1007/s00778-024-00835-2

[32] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS*. 2503–2511.

[33] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, New York, NY, USA, 1171–1188. https://doi.org/10.1145/3299869.3319863

[34] Yu Shen, Yupeng Lu, Yang Li, Yaofeng Tu, Wentao Zhang, and Bin Cui. 2022. DivBO: Diversity-aware CASH for Ensemble Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2958–2971. https://proceedings.neurips.cc/paper_files/paper/2022/hash/13b2f88be223cd2b4d6be67b56e02fa8-Abstract-Conference.html Available at https://arxiv.org/abs/2302.03255.

[35] sklearn.pipelines. 2025. Code repository. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html.

[36] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2951–2959.

[37] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 535–546. https://doi.org/10.1109/ICDE.2017.109

[38] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. 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 (KDD)*. ACM, 847–855. https://doi.org/10.1145/2487575.2487629

[39] Tanja Tornede, Alexander Tornede, Jonas Hanselle, Marcel Wever, Felix Mohr, and Eyke Hüllermeier. 2023. Towards Green Automated Machine Learning: Status Quo and Future Directions. *Journal of Artificial Intelligence Research* 77 (2023), 427–457. https://doi.org/10.1613/jair.1.14340

[40] University of Mons. 2025. Cluster website. Available at: https://www.ceci-hpc.be/clusters.html.

[41] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model

Diagnosis. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1285–1300. https://doi.org/10.1145/3183713.3196934

[42] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: A Fast and Lightweight AutoML Library. In *MLSys 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2021/hash/1ccc3bfa05cb37b917068778f3c4523a-Abstract.html

[43] Dongxin Xin, Stephan Macke, Luodong Ma, Jiannan Liu, Shiyu Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (2018), 446–460. https://doi.org/10.14778/3297753.3297763

[44] Anatoly Yakovlev, Hesam Fathi Moghadam, Ali Moharrer, Jingxiao Cai, Nikan Chavoshi, Venkatanathan Varadarajan, Sandeep R. Agrawal, Sam Idicula, Tomas Karnagel, Sanjay Jinturkar, and Nipun Agarwal. 2020. Oracle AutoML: A Fast and Predictive AutoML Pipeline. *Proceedings of the VLDB Endowment* 13 (2020), 3166–3180. https://doi.org/10.14778/3415478.3415542

[45] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 3887–3897.

[46] Lukas Zimmer, Marius Lindauer, and Frank Hutter. 2020. Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL. *arXiv preprint arXiv:2006.13799* (2020).

[47] Fatjon Zogaj, José Pablo Cambronero, Martin C. Rinard, and Jürgen Cito. 2021. Doing More with Less: Characterizing Dataset Downsampling for AutoML. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2059–2072.