Oracle AutoML: A Fast and Predictive AutoML Pipeline

Anatoly Yakovlev, Hesam Fathi Moghadam, Ali Moharrer, Jingxiao Cai,
Nikan Chavoshi, Venkatanathan Varadarajan, Sandeep R. Agrawal,
Sam Idicula*, Tomas Karnagel, Sanjay Jinturkar, Nipun Agarwal

{anatoly.y.yakovlev,hesam.fathi.moghadam,ali.m.moharrer,jingxiao.cai,nikan.chavoshi,venkatanathan.varadarajan,
sandeep.r.agrawal,tomas.karnagel,sanjay.jinturkar,nipun.agarwal}@oracle.com,*sam_idicula@hotmail.com

Oracle Labs

ABSTRACT

Machine learning (ML) is at the forefront of the rising popularity of data-driven software applications. The resulting rapid proliferation of ML technology, explosive data growth, and shortage of data science expertise have caused the industry to face increasingly challenging demands to keep up with fast-paced develop-and-deploy model lifecycles. Recent academic and industrial research efforts have started to address this problem through automated machine learning (AutoML) pipelines and have focused on model performance as the first-order design objective. We present Oracle AutoML, a novel *iteration-free* AutoML pipeline designed to not only provide accurate models, but also in a shorter runtime. We are able to achieve these objectives by eliminating the need to continuously iterate over various pipeline configurations. In our feed-forward approach, each pipeline stage makes decisions based on metalearned proxy models that can predict candidate pipeline configuration performances before building the full final model. Our approach, which builds and tunes only the best candidate pipeline, achieves better scores at a fraction of the time compared to state-of-the-art open source AutoML tools, such as H2O and Auto-sklearn. This makes Oracle AutoML a prime candidate for addressing current industry challenges.

PVLDB Reference Format:
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. Oracle AutoML: A Fast and Predictive AutoML Pipeline. *PVLDB*, 13(12): 3166-3180, 2020.
DOI: https://doi.org/10.14778/3415478.3415542

*This work was done when the author was an employee at Oracle Labs

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415542

1. INTRODUCTION

By 2025, IDC [47] predicts increasing data generation rates will usher in the age of applications with rapid model[1] build-to-use cycles. Deployed models will need fast automated retuning on fresh datasets [60] to manage drifts between trained and inference data. Such rapid model development cycles require an efficient ML pipeline, which we define as obtaining an accurate model in a short period of time. Additionally, an automated pipeline is necessary, as the usecase spans from enabling a novice user with no data science background to use machine learning, to bootstrapping a data scientist with a near optimal solution for a given dataset. Both ends of the spectrum require an AutoML pipeline to accurately generate predictions akin to a manually generated model. In an industrial setting, the ability to get results fast is crucial to jump start a new machine learning project and minimize the build-to-use time of updating an existing model.

Identifying the right model for a given dataset, which in this work we restrict to tabular classification datasets, involves selecting the best algorithm[2], the best set of rows and features, and the best algorithm hyperparameters. There are hundreds or thousands of potential combinations. Given this large search space, conventional wisdom is to have optimizers explore various pipeline configuration parameters together to effectively capture dependency among such parameters. For instance, choice of feature subsets can affect the model performance on different algorithms and vice versa. Hence, many state-of-the-art AutoML optimizers [13, 14, 20, 25, 38, 57] are iterative and need to evaluate a large number of pipeline permutations. However, this takes a long time as we will show in Section 5.1, making iterative pipelines impractical for large datasets with short time budgets.

In this work, we demonstrate how an unconventional *iteration-free* AutoML pipeline can significantly speed up the optimization process, while achieving competitive model performance relative to state-of-the-art AutoML pipelines. We take a radically new approach of rapidly narrowing the search space by doing *one-pass* preprocessing, algorithm selection, algorithm-specific adaptive data reduction, i.e., row- and feature-wise dataset sampling, and hyperparameter tuning, as shown in Figure 1.

One potential drawback of a non-iterative approach can be loss of accuracy due to omission of trials of certain pipeline configuration combinations, as will be discussed in Section 5.

[1]Throughout this paper, *model* refers to the end result of applying an ML algorithm to a dataset, with given hyperparameters.
[2]Neural architecture search and ensembles of different models are complementary to our work and are not in the scope of this paper.

Figure 1: Oracle AutoML: a non-iterative AutoML pipeline.

This can be more pronounced for longer time budgets. We minimize such risks by metalearning a proxy model per algorithm that accurately predict relative ranking between different pipeline configurations.

We make the following novel research contributions:

Iteration-free optimizer. Our iteration-free sequence of ML pipeline stages, consisting of algorithm selection, adaptive data reduction, and hyperparameter optimization, is first of its kind. Each pipeline stage optimization outcome is final and only affects the downstream stages.

Proxy Models. A metalearned set of proxy models that are predictive of relative performance of algorithms and data subsets.

Adaptive Data Reduction. Selects a representative sample of a dataset, along both the row and feature dimensions, optimized for a selected algorithm. Adaptive data reduction speeds up hyperparameter optimization with minimal impact on predictive performance of models.

HyperGD. Highly parallel gradient-based hyperparameter optimizer that performs asynchronous optimization across different hyperparameter dimensions in parallel.

These novel aspects of Oracle AutoML allow it to be significantly faster and, in most cases, more predictive than state-of-the-art AutoML approaches (Section 5.1). Our pipeline has recently been commercially released and is globally available to all Oracle customers as part of Oracle Cloud Data Science Platform [39].

In the following sections, we first provide an overview of related work in Section 2. We then follow with a high-level overview of the our pipeline in Section 3. Section 3.1 describes the process of configuring hyperparameters of our proxy model. We describe algorithm selection, adaptive data reduction, and hyperparameter optimization in Sections 3.2, 3.3, and 3.4, respectively. Section 4 describes our experiment methodology, and Section 5 provides the results and comparison of Oracle AutoML's performance relative to state-of-the-art pipelines. We share lessons we learned in Section 6 and conclude the paper in Section 7.

Commonly used abbreviations in this paper are: Adaptive Data Reduction (ADR), cross-validation (CV), our hyperparameter optimizer module (HyperGD), and Gradient-based Search Space Reduction (GrSSR).

2. RELATED WORK

End-to-end automation of machine learning has been the subject of many recent works [1, 2, 13, 16, 20, 29, 37, 38, 56, 63]. Researchers have tackled this optimization problem using several different approaches. The first approach primarily relies on Bayesian Optimization [5, 25, 57], which uses a probabilistic model to capture different hyperparameter configurations and their performance. Auto-sklearn [13], one of the most notable works relying on this approach, adopted a random-forest-based sequential model-based optimization technique [22] for general algorithm configuration. It uses metalearning to identify a previously optimized dataset closest to the given dataset and uses the known dataset's configuration to bootstrap the iterative optimization process.

A second class of solutions frame the AutoML problem as a Recommender System [16, 37, 56], where the system maintains a record of the best configuration found for each dataset it has previously encountered. Given a new dataset and the results of several initial trials, the system uses similarity to known datasets and configurations to suggest the next configurations to evaluate. Probabilistic Matrix Factorization is, typically, at the core of recommender systems [16].

A third approach relies on genetic evolutionary algorithms . One notable example is the Tree-based Pipeline Optimization Tool (TPOT) [38], which automatically optimizes a machine learning pipeline built around Scikit-Learn.

All three approaches listed above are iterative in nature, requiring multiple iterations to produce a model. Instead, we use a non-iterative approach to predict relative algorithm performance throughout the pipeline, improving efficiency compared to sequential and iterative approaches.

Additionally, these three approaches have a known issue called the *cold-start problem* - inability to predict how a pipeline configuration will perform on a new dataset. This is avoided by leveraging metalearning, where different pipeline configurations' behavior is learned on a wide variety of datasets [14, 18, 46, 67]. Other uses of metalearning utilize a known set of dataset metafeatures, such as statistical descriptions of dataset features along with some generic landmarks, such as 1NN, Naive Bayes, and PCA [43]. In Oracle AutoML, we rely on metalearning to obtain hyperparameter configurations for our proxy models. We use these proxy models in our pipeline to estimate relative performance of every algorithm and make decisions based on these estimates. This unique use of metalearned proxy models enables our highly efficient non-iterative pipeline architecture.

Several commercial AutoML products are also available for customer use. Microsoft Azure [2] uses matrix factorization and Bayesian optimization to automate selection and tuning of ML algorithms. Google's AutoML [63] exploits deep reinforcement learning and neural architecture search to find the best model. It also transfers learned knowledge to the new task. Amazon SageMaker Autopilot [1] is another AutoML platform which has components such as data preprocessing, algorithm selection, and hyperparameter tuning. Given a dataset, it picks the optimal combination of these components and uses it to train a pipeline.

Below, we explore the prior art in the individual ML pipeline stages and highlight the differences from ours.

Algorithm Selection. Past efforts on algorithm selection have generally been done in combination with hyperparameter optimization and have been based on iterative approaches such as Bayesian optimization [22], Probabilistic Matrix Factorization [16], or genetic evolutionary algorithms [38]. We use proxy models to predict tuned performance of each algorithm for the given new dataset. Their usage enables us to achieve higher efficiency than iterative solutions (Section 5).

Adaptive Data Reduction. Sampling rows on large datasets is a well-known method of speeding up ML pipelines. The main challenge with sampling is the class imbalance problem [8, 21], typically handled by doing over-, under-, or hybrid-sampling [9, 58]. Some approaches modify a model to inherently handle imbalance [11, 55]. Unlike prior work that is independent of the target algorithm, our row sampling minimizes score loss by greedily obtaining the smallest possible sample for the algorithm identified by the algorithm selection stage.

Traditional feature selection methods [7, 26, 36] rely on one of these three approaches: 1. filter methods, which make selections based on intrinsic feature properties, such as correlation to the target variable; 2. wrapper methods, which make selections based on model performance, capturing interactions between features; 3. embedded methods, which rely on a target models' own feature selection process (not all models have this capability), such as RandomForest feature importances. While each method has its own merits, no single approach performs best on all datasets. We use a novel combination of filter, wrapper, and embedded methods. This makes our feature selection more resilient to the shortcomings of a single approach across a broad spectrum of datasets. We explore feature subset sizes based on exponential growth and evaluate them using our proxy models.

Hyperparameter optimization. Among hyperparameter optimization approaches, Bayesian [5, 6, 19, 22, 23, 52] and random search based optimizers [4, 27, 54] are the most common. Bayesian optimizers usually use Gaussian Processes for global optimization of unknown functions. Their complexity is cubic with the number of trials, making them computationally expensive. Random search optimizers perform well [4, 27] but require careful guidance using other optimizers in order to be fast [12]. Gradient descent optimizers are popular and can take advantage of data-level parallelism [30, 31, 45, 66], but gradients for hyperparameters are not well defined. Coordinate descent optimizes one dimension of hyperparameter search at a time and is parallelizable across the dimensions [33, 41]. Our hyperparameter tuning algorithm (HyperGD) is similar to coordinate descent because we tune across different dimensions in parallel. However, we bootstrap our search with multiple points across the dimensions, and, more importantly, utilize rapid search space reduction based on gradient intersections. Finally, we perform gradient descent as the last step (Section 3.4).

3. ORACLE AUTOML

We formally define the AutoML pipeline optimization problem similarly to the Combined Algorithm Selection and Hyperparameter optimization [57], but include Adaptive Data Reduction (ADR) as another dimension. Given a dataset D_{train} with N samples and K features, the aim is to find the combination of the best algorithm $A^\star$, data sample $D^\star_{train}$ and hyperparameter setting $\lambda^\star$ by minimizing the average loss function $\mathcal{L}$, where $\mathcal{L}$ is any user-defined misclassification rate, and is logloss in this paper, as explained in Section 4,

$$D^\star_{train}, A^\star, \lambda^\star \in \underset{\substack{n\in N, k\in K,\\ A^{(j)}\in\mathcal{A},\\ \lambda\in\Lambda^{(j)}}}{\operatorname{argmin}} \mathcal{L}\big(A^{(j)}_\lambda, D^{(n,k)}_{train}\big) \qquad (1)$$

We introduce several novel contributions in the design of Oracle AutoML, which produce *fast and accurate* results. First, we implement the optimizer as a sequential non-iterative architecture. This design choice significantly speeds up the pipeline as every stage's decision is made in a feed-forward manner.

Our pipeline consists of a predefined set of stages (Figure 1), starting with data preprocessing. We implement commonly utilized preprocessing steps [24], including missing value imputation, label encoding, and normalization.

The next stage consists of algorithm selection, which determines the best algorithm ($A^\star$) for this dataset. Because subsequent stages depend on an underlying algorithm, this stage eliminates the need for iterative optimization. Since algorithm selection is pivotal to the performance of the entire pipeline, we rely on carefully crafted proxy models to select the best algorithm for the dataset. These proxy models act as indicators of how well a given algorithm will perform on the dataset of interest. Their highly predictive nature (Section 5.2) helps us mitigate score degradation, which would normally be associated with a non-iterative pipeline.

With knowledge of the best algorithm ($A^\star$), ADR aims to reduce the number of dataset rows and select a subset of features without compromising model performance ($D^\star_{train}$). Both row sampling and feature selection rely on proxy models to score samples and subsets.

HyperGD (hyperparameter optimization) is the final stage of the pipeline and it aims to fine tune the selected algorithm's hyperparameters ($\lambda^\star$). It is the most time-consuming stage of our pipeline and it benefits from ADR (Section 5.4). Additionally, all stages of our pipeline are parallelized wherever possible. For instance, all per-algorithm, per-feature, and per-hyperparameter computations are executed in parallel.

To make Oracle AutoML more robust, we have also introduced a time budget feature to prevent our optimizer from terminating without producing a tuned model. We accomplish this by respecting a time budget argument at every stage of the pipeline. For very short time budgets or large datasets, not all pipeline stages may have a chance to fully execute. Therefore, we implement a fallback strategy to ensure our pipeline produces a tuned model, regardless of which pipeline stage the time budget is exhausted. First, if the time budget is exhausted before algorithm selection is completed, we default to NaiveBayes proxy model as the tuned model. Second, if the budget is exhausted during ADR, the dataset sample with the highest cross-validation score will be used. Specifically, this has an effect on the selected features, which will be used for the final model. If the budget is exhausted before HyperGD stage is reached, Oracle AutoML outputs a proxy model corresponding to the selected algorithm as the tuned model. Finally, if time budget is exhausted during the hyperparameter optimization stage, we select the best tuned model so far, based on the maximum cross-validation score.

3.1 Proxy Models

A key requirement for Oracle AutoML is to make fast accurate decisions during algorithm selection, data reduction, and hyperparameter optimization. A wrong decision, especially at the algorithm selection stage, could severely affect the downstream stages and the resulting pipeline configuration.

Proxy models are performance predictors that we use in all stages to make our pipeline iteration-free. For the proxy models to function this way, they need to satisfy the following requirements: (a) be an instance of the ML algorithm, whose best performance we want to predict; (b) have relative performance representative of the tuned model, without requiring hyperparameter tuning; (c) be able to accurately predict relative ranking of different dataset subsets.

Given such a proxy model, for example, we could pick the best feature subset out of K different subsets by just ranking their scores with the proxy model. The primary challenge is to find a single proxy model per ML algorithm that is

predictive for any never-before-seen dataset. We leverage *metalearning* to identify these proxy models by observing each algorithm's behavior on a wide variety of datasets and hyperparameters. This involves three steps: (a) generate large number of representative hyperparameter variations per algorithm (i.e., candidate proxy models), (b) evaluate each model's performance on a wide variety of datasets, and (c) heuristically identify a proxy model per algorithm from all candidates.

Generating candidate proxy models. We explore the hyperparameter space for each ML algorithm using the following methodology:

1. For each hyperparameter we define a search range; numerical parameters have generous upper and lower bounds, and all values of categorical parameters are considered.
2. We generate R random hyperparameter choices, where each hyperparameter value is obtained by uniformly sampling at random from its range.
3. We generate an additional M hyperparameter choices by fixing each hyperparameter to its default value (determined by algorithm developers [40]), and randomly sample the other hyperparameters uniformly from their respective ranges.
4. These $R + M$ hyperparameter choices constitute a candidate proxy model pool. For each of these candidates, we perform K-fold cross-validation (CV) and measure mean CV score. This is repeated for D datasets.

The above approach results in an $(R+M) \times D$ metadataset for each algorithm. Usually, algorithm developers and data scientists pick defaults for their algorithms so that they perform well on a variety of datasets. We pursue the same goal with our proxy models. We use these defaults in our search space, as well as random points around them to be immune to any human error. This process is done only once per new algorithm, when it is added to Oracle AutoML's search space. In this paper, we only consider datasets in our separate training corpus to avoid data leakage during proxy model search.

Heuristic to identify proxy models. We take a systematic approach to selecting proxy model hyperparameters, consisting of the following steps for each algorithm:

1. For every dataset, identify the best hyperparameter configuration based on the highest CV score.
2. For every dataset, subtract the highest CV score from the CV score of every hyperparameter configuration to obtain the corresponding configuration's *score difference* value.
3. For every hyperparameter configuration, average the score difference values across all datasets to obtain *average score difference* value.
4. Pick the hyperparameter configuration with the lowest average score difference.

Each stage of the Oracle AutoML pipeline uses these proxy models to efficiently narrow down the search space. Algorithm selection uses them to rank algorithms, ADR uses them to identify the relevant segment of the dataset, and HyperGD uses them to bootstrap optimization. Table 1 summarizes the explored hyperparameter ranges, total number of configurations evaluated for every algorithm, and the selected hyperparameters for each proxy model.

Table 1: Hyperparameter ranges, number of evaluations, and selected proxy models per algorithm.

Algo	Hyperparam	Range	Evals	Selected
AdaBoost	learning_rate	[1e-3, 2]	249	0.0513
	n_estimators	[5, 500]		50
Decision Tree	min_samples_leaf	[1e-8, 0.5]	200	0.2527
	max_features	[1e-8, 1]		0.9638
	min_samples_split	[1e-8, 1]		0.9638
	class_weight	[off, bal]		off
Extra Trees	min_samples_leaf	[1e-8, 0.5]	232	5.96e-5
	criterion	[ent, gini]		gini
	min_samples_split	[1e-8, 1]		0.0314
	max_features	[1e-8, 1]		0.4839
	n_estimators	[5, 500]		66
	class_weight	[off, bal]		off
KNN	weights	[uni, dis]	92	uniform
	n_neighbors	[2, 32]		31
Lin. SVC	C	[3e-2, 512]	87	13.78
	class_weight	[off, bal]		off
Logistic Regres.	solver	[n-cg, lbfgs, libl, sag]	92	lbfgs
	C	[3e-2, 512]		1.0
	class_weight	[off, bal]		off
LGBM	num_leaves	[3, 750]	234	509
	boosting_type	[gbdt, dart, goss]		dart
	learning_rate	[1e-4, 1]		0.2711
	min_child_weight	[0, 20]		0.001
	max_depth	[1, 10]		5
	reg_alpha	[1e-10, 1]		0.7667
	reg_lambda	[1e-10, 1]		0.9864
	n_estimators	[5, 500]		294
	class_weight	[off, bal]		off
Random Forest	max_features	[1e-8, 1]	220	0.3725
	min_samples_split	[1e-8, 1]		5.43e-5
	n_estimators	[5, 500]		283
	class_weight	[off, bal]		off
SVC	C	[3e-2, 512]	135	305.97
	gamma	[3e-5, 8]		0.0077
	class_weight	[off, bal]		balanced
XGBoost	reg_alpha	[0, 4]	272	1.0581
	reg_lambda	[0, 100]		17.57
	booster	[gbt, dart]		gbtree
	min_child_weight	[0, 20]		1
	learning_rate	[1e-4, 1]		0.3502
	max_depth	[2, 10]		6
	n_estimators	[50, 500]		95
	class_weight	[off, bal]		off

3.2 Algorithm Selection

Algorithm selection can have a significant impact on the achieved score, and the only way to guarantee optimality is to *exhaustively* train every algorithm on the given dataset [48]. State-of-the-art approaches, such as Auto-sklearn [13] and TPOT [38], treat an algorithm as another hyperparameter in a large search space. This makes exploration costly and increases the time to produce a reasonable solution, negatively impacting user experience. We model automatic algorithm selection as a *score prediction* problem. We use our per-algorithm proxy models to rank algorithms for a given dataset, narrowing down the search space for subsequent stages of the pipeline to a single algorithm.

The proxy models provide an indication of the tuned score of the dataset on each algorithm relative to other algorithms. Figure 2 shows a block diagram of our approach. The proxy models for all algorithms are executed in parallel and the average cross-validation score is used to rank the available algorithms $\mathcal{A}$. The goal of algorithm selection is to identify a relative ranking among algorithms and the actual scores are unimportant. To further reduce the runtime of this stage for large datasets, we sample D'_{train} on a per class basis, where *each target class* is limited to 50K samples. Based

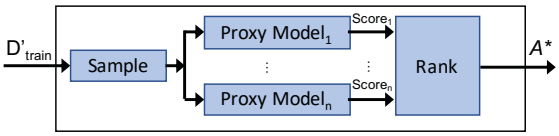


Figure 2: Algorithm selection block diagram, consisting of dataset sampling, proxy model evaluation, and ranking of obtained CV scores, used to select the best algorithm A^*.

on train corpus evaluations, we have empirically found this limit provides a reasonable tradeoff between runtime and accuracy of ranking algorithms. The best algorithm (A^*), corresponding to the proxy model that produces the highest score, is passed onto the next stage.

3.3 Adaptive Data Reduction

Dataset reduction by means of sampling rows and columns (commonly referred to as feature selection) can significantly reduce the runtime of the following hyperparameter optimization stage. Furthermore, a sample that is statistically representative of the original dataset will have minimal adverse impact on model score. We present a novel sampling approach to generate the most efficient sample of the dataset, for the algorithm (A^*) chosen by algorithm selection.

3.3.1 Row Sampling

The goal of this stage is to select a subset of dataset rows that is representative of the original dataset. We do this by climbing the dataset learning curve. The dataset is sampled iteratively from a small subset to the full dataset size and each sample is scored by the proxy model (P^*), representing the algorithm (A^*) selected by algorithm selection. The goal is to find the smallest sample size of a dataset, for use in subsequent pipeline stages, without sacrificing model quality. Consecutive sample sizes, with increasing size, are tried until a score tolerance threshold is met. Unsampled dataset is used if all the sample sizes are exhausted without meeting the threshold. Figure 3 shows the row sampling procedure.

In Oracle AutoML, we default the lower bound of the sampling range to 1000 samples per class to generate a sample with a distribution that is representative of the original dataset (for binary classification ∼3% margin of error with 95% confidence) [10]. We sample each class independently to avoid class imbalance and if any class has fewer samples than the lower bound, we do not sample that class. The upper bound of the sampling search range is limited to the maximum number of samples in the majority class. Finally, the sample sizes in-between the lower and upper bound are determined by a combination of linearly and geometrically spaced values between these bounds, resulting in more sample sizes at the lower end of the range. This is preferable, as we would like to look more aggressively for samples on the lower end to optimize runtime. We cap the total trials to 10 sample sizes, comprising of the lower bound, four linearly spaced, four geometrically spaced, and the upper bound, sorted from smallest to largest. A larger number of samples can lead to quicker convergence; however, it can also increase the chances of getting stuck in a local minimum.

3.3.2 Feature Selection

The goal of feature selection is to find a subset of dataset features that are representative of the original dataset. Our approach is a combination of filter, wrapper, and embedded

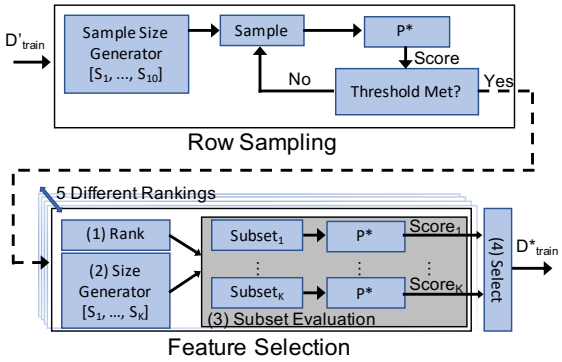


Figure 3: Adaptive data reduction performs row sampling followed by feature selection on a given dataset. P^* refers to the proxy model for the selected algorithm A^*.

methods [36]. In particular, we first rank features, then evaluate subsets of these features on the provided proxy model (P^*). The number of evaluations performed by this stage is typically much greater than the row sampling stage. Hence, all subsets are evaluated in parallel, in contrast to the sequential implementation in the row sampling scheme described previously.

Feature selection can be split into 4 main steps as depicted in Figure 3; (1) feature ranking, (2) subset size generation, (3) subset evaluation, and (4) subset selection. Feature ranking is the procedure by which features are ordered by their importance. We use multiple ranking algorithms from [40] in order to better generalize across a wide variety of datasets: (a) mutual information between every feature and the target, (b) ANOVA F-value, (c) an ensemble based method (feature importances from Random Forests model), (d) a boosting based method (feature importances from AdaBoost model), and (e) the average of the normalized values from (a-d). Rankings (a-d) return values representing correlation or importance of each feature with respect to the target, where higher value means the feature is more important. To obtain ranking (e), we first normalize the returned values for rankings (a-d) and average the normalized values per feature to obtain a new ranking that incorporates all four rankings. These five rankings sufficiently capture the diversity across a wide range of different algorithm types and datasets.

Subset size generation is the procedure by which we choose the number of features that are selected for evaluation. The subset exploration space can be vast, as there are 2^K different potential feature combinations (K is the total number of features). In our approach, we reduce this search space by ranking the features, and selecting only the top (k) number of features for evaluation. If one were to incrementally add one feature to the evaluation subset based on decreasing rank priority, there would be a total of K subset sizes that are evaluated. In order to speed up the process, we use an exponential growth function with a growth factor of 20%. This reduces the number of subset evaluations per ranking algorithm to be proportional to $ln(K)$. Furthermore, the exponential growth biases the subset sizes towards the smaller end. This outcome is desirable because smaller subsets are faster to evaluate and the ultimate goal is to find the smallest subset that does not degrade score.

Subset evaluation is the procedure by which the chosen feature subsets are evaluated on the proxy model using the row sampled dataset. As there are five rankings, the number

of subsets that are evaluated is $\sim 5ln(K)$. As an optimization, we keep track of all the evaluated subsets and do not repeat evaluations. This is significant as different rankings may produce overlapping ranking orders. These proxy model evaluations are performed in parallel.

Subset selection is the procedure by which the feature subset that produced the highest score on the proxy model is selected. If desired, the user has the ability to trade-off score for speed by selecting a smaller subset size. This is implemented using a score tolerance threshold variable that can be passed to this stage. The default threshold is zero to minimize score degradation due to this stage.

3.4 HyperGD

Hyperparameter optimization is a well-studied problem. However, existing approaches have little emphasis on optimizing for both time and available resources together [28, 53], especially when tuning for medium to large datasets. Note reducing the number of model evaluations (or trials) only partly improves efficiency as the cost of these trials may vary significantly. In the context of AutoML efficiency, we define cost as time taken for model evaluation. Further, many optimization algorithms [19, 22] perform best when the search is done sequentially. Such an approach is often prohibitively expensive in terms of runtime [42, 53]. Hyperparameter optimization is the most time-consuming stage of Oracle AutoML, as shown in Section 5.1. Therefore, in *HyperGD*, we optimized for efficient resource utilization via parallelization without compromising model performance.

3.4.1 HyperGD: A Parallel, Gradient-based Hyperparameter Optimizer

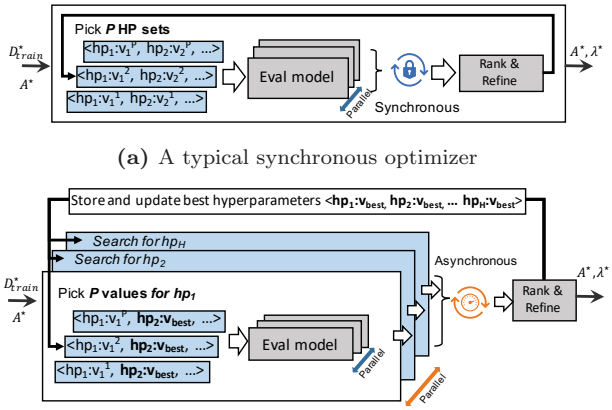
A typical hyperparameter optimizer ([22]) selects and evaluates a batch of hyperparameters, waits for all the evaluations to complete, before selecting the next batch of hyperparameter values based on the results of the current batch. Each of these evaluations is called a *trial*, and each trial takes arbitrarily long, depending on the dataset and choice of hyperparameters. Figure 4a shows the process. The Rank & Refine block typically uses a Bayesian algorithm [50].

The primary problem with such a synchronous parallel algorithm is poor resource utilization. Trial-to-trial evaluation time can differ by orders of magnitude as shown in Figure 5. This results in stragglers that poorly utilize multiple compute units (cores or nodes) as well as block progress to the next iteration. Hyperparameter optimizers, by design, evaluate different hyperparameters within a batch, making stragglers unavoidable. The stragglers need to be mitigated in order to efficiently utilize all available resources.

In contrast, HyperGD is a highly parallel and asynchronous algorithm that parallelizes trials during search of a given hyperparameter as well as trials across other hyperparameters as shown in Figure 4b. We achieve this high-degree of parallelism by asynchronously gathering and using the best hyperparameters from all completed trials whenever launching any new trial. Further, we do not wait for all the results to complete from a batch of model evaluations. Both of these optimizations are possible because of the novel Gradient-based Search Space Reduction (GrSSR) in HyperGD.

3.4.2 Gradient-based Search Space Reduction

The goal of the GrSSR algorithm is to rapidly narrow the search space for each hyperparameter, given a wide



(a) A typical synchronous optimizer

(b) Asynchronous parallel HyperGD optimizer

Figure 4: Synchronous versus asynchronous hyperparameter optimizer block diagram. Key advantage of asynchronous optimizer is the ability to update best value for each hyperparameter without waiting for the entire batch to complete, resulting in significant speedup.

search range (see ranges in Table 1). For example, with only 10 choices per numerical hyperparameter, a grid search for SVC and XGBoost would result in 400 and 4M choices, respectively. A sample visualization of the optimization process, while tuning the gamma hyperparameter on SVC, is shown in Figure 6. For ease of explanation, let us assume a simple 1-D search space, where x-axis is the value for a given hyperparameter, and y-axis is the objective error metric (lower is better). With the goal of narrowing the initial search range towards the minimum of the error curve, we pick P point-pairs to estimate the gradients at these points. So, if we selected P points $v_i^1, v_i^2, ..., v_i^P$ per hyperparameter hp_i, each point is matched with another point in its ϵ_i neighborhood. For each hyperparameter, ϵ_i is selected relative to the initial range of search space.

Next, the direction of the minimum is estimated by finding the intersection point of the top two pairs' gradients. Here, the best two pairs must contain at least the top three trials (lowest error) in the batch. We, then, narrow the search range by picking the next P point-pairs in the vicinity of the intersection of the two gradients. The points are chosen to be logarithmically spaced, with points more densely spaced around the intersection.

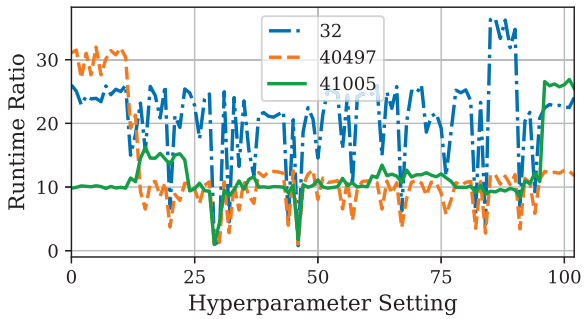


Figure 5: SVC fit time across the same set of $\sim$ 100 hyperparameter settings for three different datasets from our train corpus. The fit time for each setting is normalized to the fit time with default hyperparameter settings. We can see up to $\sim 40\times$ variation in fit time.

In this algorithm, we make two approximations that are vital to parallelize and speed up the hyperparameter optimization process.

1. We assume hyperparameters can be optimized independently of each other. There are dependencies among some hyperparameters, whose values depend on other hyperparameters' values [34]. However, this is a well understood and accepted optimization technique [33, 41]. This simplifying assumption allows us to completely parallelize the search across all hyperparameter dimensions without any synchronization. Each search eagerly updates the other hyperparameter searches with its best hyperparameter value found so far, asynchronously. This phase of the algorithm mimics the behavior of coordinate descent algorithm [64] with asynchronous updates to individual hyperparameters.

2. In addition, we also eagerly reduce the search space where we only wait for $P_{min} < P$ point-pairs out of all trials in the current batch. Theoretically, we need a minimum of two point pairs to compute the point of intersection. In our search, we wait for 3 point pairs from the current batch in order to identify the best two pairs (so far) for GrSSR algorithm. The pending trials help refine the search space further when they complete.

Note GrSSR algorithm handles few corner cases that are not addressed here. For instance, two gradients under consideration may be parallel (non-intersecting) and hence would fail to point to the direction of the potential minimum. In such cases, we either wait for more trials, or select the current best value and give up the search on that particular hyperparameter. This is because a lack of gradient indicates the algorithm is insensitive to variations of this particular hyperparameter or the search space is too narrow.

Finally, when the search space cannot be further narrowed, we use gradient descent to fine-tune the hyperparameter values. As gradient descent is generally slow and sequential, we perform a short five-epoch descent with a learning rate of 0.1 to descend on the last-mile of the error score curve. We do gradient descent independently for every hyperparameter. This two stage process of GrSSR and gradient descent constitutes our novel *HyperGD* algorithm.

Apart from the highly parallel and asynchronous nature of the algorithm, there are a few other optimizations responsible for HyperGD's efficiency.

Bootstrapping Search. Initial choices of hyperparameters are critical to efficiently bootstrap the search for any black-box optimizer. For our HyperGD algorithm, we vary each hyperparameter $2 \times P$ ways (P point-pairs). In order to limit the chances of getting stuck in a local-minimum, we pick P points $v_i^1, v_i^2, ..., v_i^P$ per hyperparameter hp_i that are linearly-spaced within the predefined search range. As discussed above, each point has a matching pair within its ϵ_i neighborhood. We repeat this for H hyperparameters. To fix other hyperparameter values, while varying one, we use predefined defaults for other hyperparameters, called bootstrap defaults $< hp_1 : g_1, hp_2 : g_2, ..., hp_H : g_H >$. Hence, we arrive at $2 \times P \times H$ hyperparameter sets by using predefined bootstrap defaults as constants for the non-varying hyperparameters. For instance, for hyperparameter hp_1, the $2 \times P$ hyperparameter sets to be tried are: $< hp_1 : v_1^1, hp_2 : g_2, ..., hp_H : g_H >$, $< hp_1 : v_1^1 + \epsilon_1, hp_2 : g_2, ..., hp_H : g_H >$ $< hp_1 : v_1^2, hp_2 : g_2, ..., hp_H : g_H >$, $< hp_1 : v_1^2 + \epsilon_1, hp_2 : g_2, ..., hp_H : g_H >$, ..., $< hp_1 : v_1^P, hp_2 : g_2, ..., hp_H : g_H >$, , $< hp_1 : v_1^P + \epsilon_1, hp_2 : g_2, ..., hp_H : g_H >$.

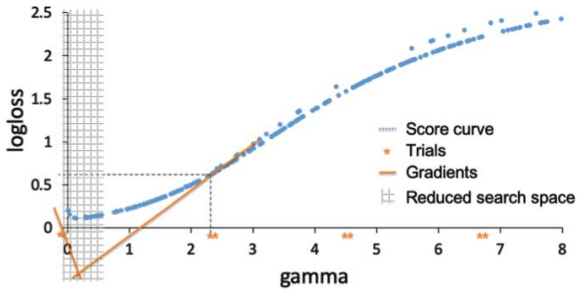


Figure 6: Sample visualization of gradient-based search space reduction algorithm in action, while tuning the `gamma` hyperparameter of an SVC model on a real-world dataset. Logloss error is plotted on the y-axis.

We use two bootstrap defaults: (a) our novel algorithm specific proxy model hyperparameters (P^*) and (b) the lowest edge of the search space range. The first hyperparameter set ensures we explore a wider search space around informed defaults. The second hyperparameter set selects points in the lower end of each hyperparameter plane, so that our search includes points on either side of the potential minimum. In addition, we also found exploring this part of the search space to be faster for most algorithms. Overall, we bootstrap the search with all $2 \times P \times H$ trials in parallel.

Handling Categorical Hyperparameters. Until now, we have described how HyperGD handles continuous and discrete hyperparameters, for which gradients can be computed. This is not possible for categorical hyperparameters, for which there are no relative ordering between individual hyperparameter choices. An example of such a hyperparameter is the type of solver to be used in Logistic Regression, which can take values like `lbfgs`, `liblinear`, etc. (Table 1). For such hyperparameters, instead of our GrSSR algorithm, we try all variations at intervals, when values for any other numerical hyperparameters vary by more than a threshold (5%). This threshold aims to strike a balance between the cost of re-exploring different values for the categorical hyperparameters versus missing better models in the newly narrowed search space of the numerical hyperparameters.

Runtime cost awareness. In order to mitigate the impact of stragglers caused by wide variations in runtimes of each trial, we use a simple linear cost-model for each hyperparameter to pick more points toward the least-expensive side of the range. This helps reduce stragglers by limiting the variation in time taken by trials for a given batch.

4. EXPERIMENT METHODOLOGY

We outline our experiment methodology by providing details on the train and test corpora, model evaluation, scoring metric, setup, ML algorithms considered, AutoML comparisons, random states evaluated, and result visualizations.

Train Corpus. We have used a set of 130 datasets from publicly available OpenML datasets [61]. The shape distribution of these datasets is shown in Figure 7. Our proxy models are learned using only these datasets.

Test Corpus. We believe it is critical to evaluate AutoML pipelines with a diverse set of datasets with varying shapes. We randomly select 30 *new* OpenML datasets, that do *not overlap* with the train corpus. The distribution of dataset shapes is shown in Figure 7. We categorize these datasets into small, medium, and large, based on the number of

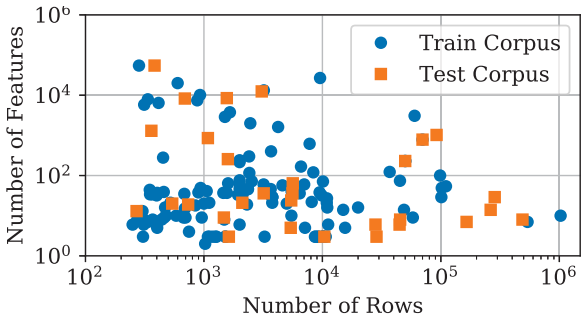


Figure 7: Diversity in dataset shapes in the training and leave-out test datasets.

data points (*samples* × *features*), as shown in Table 2. In the table, *Missing* column indicates if the dataset has any missing values in its features. We emphasize that none of the datasets in our test corpus overlap with the train corpus, to avoid data leakage. Hence, the test corpus datasets are strictly used for evaluation only.

Model Evaluation. We perform an 80-20 train-test split on every dataset in the test corpus and use the same 80% train set for training and 20% leave-out test set for final evaluations in all experiments and AutoML comparisons in Section 5. Five-fold CV is used in all pipeline optimizations.

Scoring Metric. We use the log-likelihood of the true labels for a given probabilistic classifiers predictions as the scoring metric for evaluation. In other words, *logloss* [49], is the objective function used for all model evaluations, where lower logloss is better. This metric is used in many benchmarking studies[17, 44] for comparing AutoML pipelines on a wide variety of datasets, and is a good metric [15, 62] as it punishes highly confident wrong predictions.

Setup. Experiments are run on Oracle Cloud Infrastructure using a homogeneous set of 2-socket Intel Xeon E5-2699v3 Haswell machines (36 total cores). Python 3.6.8, scikit-learn v0.20.2, numpy v1.16.2, and scipy v1.2.1 were used.

Machine Learning Algorithms. In this paper, Oracle AutoML supports the following algorithms: AdaBoost, DecisionTree, ExtraTrees, KNeighbors, LinearSVM, LogisticRegression, RandomForest, SVM, GaussianNB from scikit-learn [51], XGBoost (XGB) v0.81 from [65], and LightGBM (LGBM) v2.2.3 from [32].

AutoML Comparisons. We compare our pipeline with recent versions of Auto-sklearn(0.6.0) and H2O (3.26.0.10) because they are commonly cited, state-of-the-art [3, 59], and have readily available open-source implementations. We use the same test corpus (Table 2) for all pipelines, with the exact same train and leave-out test sets of each dataset. Of note, H2O includes neural network algorithms and Auto-sklearn builds ensembles for their final models. We do not change any of the default settings during our evaluation to prevent any negative effects on these two pipelines.

Random State. AutoML pipeline comparison experiments (Section 5.1) are based on 5 different random states, with seeds 7, 11, 13, 17, and 19. Thus, the results and visualizations include $5 * 30 = 150$ total runs per pipeline. For all other results, a single run with seed 7 is presented.

Visualizations. In order to visualize experiment results, we have elected to use box and whisker plots to summarize the result statistics. The whisker reach, which sets the length of the whiskers above and below the upper quartile and lower quartile, is kept at the default value of 1.5 times the inter-quartile range [35]. We use ratios of scores and runtimes per dataset when comparing contributions of individual Oracle AutoML stages. Intuitively, score and runtime ratios provide relative score improvement and speedup, respectively, while removing any skew introduced by averaging absolute values.

Table 2: OpenML classification leaveout test datasets.

	ID	Name	Rows	Cols.	Class	Missing
Small	3	kr-vs-kp	2556	36	2	No
	23	cmc	1178	9	3	No
	53	heart-statlog	216	13	2	No
	188	eucalyptus	588	19	5	Yes
	1067	kc1	1687	21	2	No
	1467	climate-mdl	432	20	2	No
	1489	phoneme	4323	5	2	No
	1528	volcanoes-a2	1298	3	5	No
	1533	volcanoes-b3	8308	3	5	No
	1537	volcanoes-c1	22900	3	5	No
Medium	28	optdigits	4496	64	10	No
	151	electricity	36249	8	2	No
	1069	pc2	4471	36	2	No
	1468	cnae-9	864	856	9	No
	1481	kr-vs-k	22444	6	18	No
	1483	ldpa	131888	7	11	No
	1497	wall-rbt-nav	4364	24	4	No
	1501	semeion	1274	256	10	No
	1514	micro-mass	288	1300	10	No
	41027	jungle-chess	35855	6	3	No
Large	351	codrna	390852	8	2	No
	383	tr45.wc	552	8261	10	No
	393	la2s.wc	2460	12432	6	No
	398	wap.wc	1248	8460	20	No
	554	mnist-784	56000	784	10	No
	1088	varCancers	306	54675	9	No
	1111	KDDCup09	40000	230	2	Yes
	1503	spoken-arabic	210604	14	10	No
	1597	creditcard	227845	29	2	No
	40923	Devnagari	73600	1024	46	No

5. EVALUATION OF ORACLE AUTOML

We evaluate an AutoML optimizer using two main metrics: efficiency and robustness. The goal of our evaluation is to answer the following questions:

Comparison with state-of-the-art. How competitive is Oracle AutoML compared to existing state-of-the-art AutoML, in terms of model score and speed?

Proxy model effectiveness. Do the proxy models justify iteration-free architecture without loss in score?

Benefits of each stage in Oracle AutoML. What is the contribution of each pipeline stage to overall performance, based on score and speed tradeoff?

First, we present a quantitative comparison of Oracle AutoML with several state-of-the-art open-source alternatives. Second, we look at proxy models and evaluate our methodology of identifying good proxy models for different algorithms. Finally, given the fully decomposable nature of our pipeline, we systematically disable its different stages and evaluate their impact on score and speed.

5.1 Comparison with AutoML Alternatives

We compare Oracle AutoML pipeline against two state-of-the-art open-source packages: Auto-sklearn[13] and H2O [20]. We use logloss metric to measure model performance, where lower logloss is better. Section 4 describes the methodology in greater detail.

Specifically, our comparison criteria are efficiency and robustness. For efficiency, we compare the test scores of each pipeline for a given time constraint. Each optimizer can choose to utilize their entire time budget (Auto-sklearn), or converge on a solution earlier (H2O, Oracle AutoML). For robustness, we compare the number of times each pipeline fails to produce a model for a given time constraint.

Efficiency. To compare efficiency across the AutoML pipelines, we measure the average ranking of the three pipelines with 11 different time budgets of 1, 3, 5, 7, 10, 15, 20, 25, 30, 45, 60 minutes and five different seeds over the 30 dataset test corpus. We rank each AutoML system per run (given time budget and seed) based on the leave-out test set score of each dataset, and average the ranks across datasets per run similar to [13]. Incomplete runs are assigned the worst rank of three. The result is shown in Figure 8, where rank one is the best. H2O performs significantly better than Auto-sklearn across all the runs, and Oracle AutoML is the most efficient out of all three. The fluctuations in rankings within the first 10 minutes are mainly due to Auto-sklearn and H2O not producing a model for some datasets (Table 3) within those time budgets.

These ranking plots only show the efficiency part of the story. We look at the raw performance of the pipelines by plotting the distribution of tuned model performance given a full hour to optimize (shown in Figure 9). Overall, Oracle AutoML has a very tight distribution of logloss with an average logloss (depicted by $\times$ in the figure) of 0.338, that is much better than the other two, corroborating the ranking plot. Interestingly, when comparing Auto-sklearn and H2O, the distribution is only marginally different from each other. But in fact, the average logloss for H2O is 1.882, which is much worse than the 1.014 that Auto-sklearn achieves. This is primarily because H2O seems to perform very poorly for multiple datasets (shown by the many outliers), while performing significantly better on other datasets.

Even with the maximum time budget of one hour, H2O and Auto-sklearn are not able to achieve better results for most datasets. On the other hand, our pipeline is able to quickly converge on a solution, finishing far ahead of the allotted time budget in most cases, as can be seen from the runtime breakdown across all test datasets in Figure 10. In the figure, only 5 out of 30 datasets need the full time budget of 60 minutes. Our pipeline completes optimization for majority of datasets within 500 seconds. This gives Oracle AutoML a speedup of 3.57× over H2O and 4.11× over Auto-sklearn, when aggregating runtimes across the test

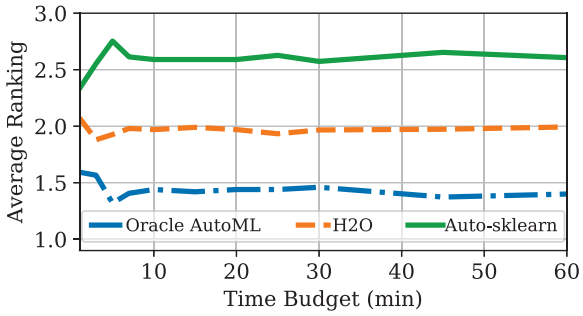


Figure 8: Average ranking across 30 datasets and 5 random seeds for Auto-sklearn, H2O, and Oracle AutoML plotted versus varying time budget. Rank 1 is the best.

Table 3: The number of evaluation trials that each AutoML framework completed for a given time budget. The expected completion is 150 because there are 30 datasets and 5 different random seeds.

Pipelines	Time budget (minutes) 1	5	10	30	60
Oracle AutoML	100%	100%	100%	100%	100%
Auto-sklearn	100%	93.3%	92%	100%	100%
H2O	78%	94.7%	96.7%	96.7%	100%

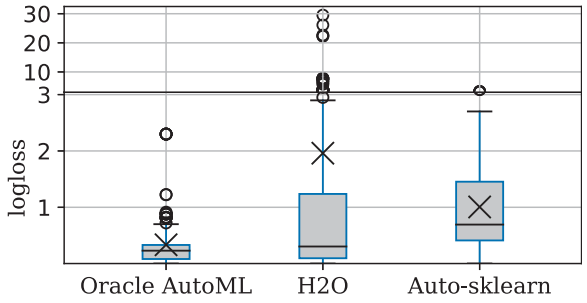


Figure 9: Comparing Oracle AutoML to H2O and Auto-sklearn with 60-minute time budget on 30 datasets. Reported numbers are average of 5 runs each with different random seed. The y-axis shows the logloss error (lower is better).

corpus. On average, across test datasets, algorithm selection, ADR, and HyperGD stages take 21%, 34%, and 45% of our pipeline's runtime, respectively. Even when iteration free, each stage adapts and optimizes to the given dataset differently, taking different times.

Oracle AutoML's efficiency benefits, primarily, stem from the performance on large datasets for which other AutoML pipelines struggle to optimize under the same time budget. The primary contributor is our quick and accurate algorithm selection using proxy models. The distribution of selected algorithms across all the evaluated runs is shown in Figure 11, where the top three selected algorithms are XGBoost, LightGBM, and LogisticRegression. As can be seen from the figure, algorithm selection is an essential stage in our AutoML pipeline, based on the diverse variety of algorithms that were selected across all trials. The non-iterative nature of Oracle AutoML and reliance on proxy models to accurately select the best algorithm for every new dataset, at the first pipeline stage, is what enables us to significantly outperform iterative optimizers.

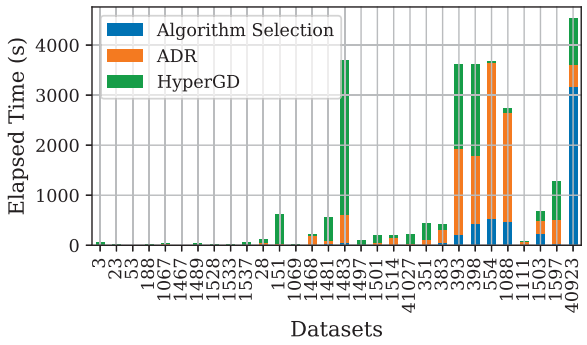


Figure 10: Runtime breakdown of stages in Oracle AutoML across our test corpus for the 60 minute time budget case.

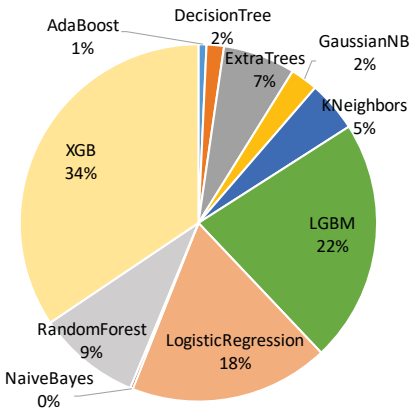


Figure 11: Distribution of selected algorithms across a total of 1650 Oracle AutoML runs corresponding to our test corpus of 30 datasets, 5 random seeds, and 11 time budgets.

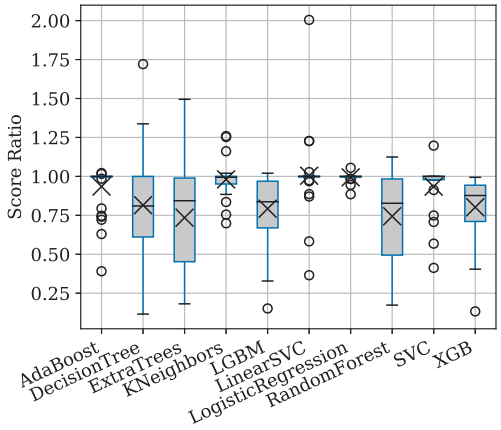


Figure 12: Logloss score of proxy models normalized to corresponding extensive search best model, for 30 test datasets across all algorithms. Normalized score ratio >1 is better.

Almost all algorithms are picked for some runs, and no one algorithm is best for all scenarios. It is worth noting, our proxy model-based algorithm selection may trade off accuracy for speed for some datasets. For instance, Oracle AutoML selects XGBoost for dataset 383 and achieves a logloss score of 0.17 losing out to H2O's 0.08. However, a score of 0.06 could have been achieved with LightGBM model. Therefore, iterative approaches can, at times, achieve better scores at the cost of longer runtime.

The second contribution to efficiency in our pipeline is the ADR stage that provides runtime reduction by rapidly selecting a small and representative sample of the dataset for HyperGD. Specifically, ADR resulted in an average of 61% reduction in dataset size (i.e., rows and features) across all datasets within a 60-minute time budget.

Robustness. We find that not all AutoML pipelines produce a tuned model within the provided time budget. Table 3 shows a subset of time budgets and, understandably, not all are optimized for smaller time budgets. For example, H2O does not complete for very small time budgets, whereas Auto-sklearn handles even one minute time budgets well. But there are cases, when even a reasonably long time budgets of 10-30 minutes result in fewer completions. For fairness to other pipelines, the results presented here are taken after our team has spent several weeks resolving any simple failures. In contrast, Oracle AutoML always terminates with a useful model given any time budget. Note that all these pipelines do not strictly adhere to the time budget and we observe, in some cases, they exceed by a few minutes, at most.

5.2 Proxy Models

In this section, we evaluate the performance of our proxy models by comparing it to model performance achieved by extensive random search of hyperparameters, default scikit-learn models [40], and by evaluating its sensitivity to the number of train corpus datasets used to learn their hyperparameters. The extensive search in Figure 12 finds the best hyperparameters by exploring 87 to 272 configurations for each dataset, depending on the algorithm. As expected, proxy models are not a replacement for hyperparameter tuning. They perform worse than extensive search on most algorithms, especially for tree-based algorithms. However, the proxy model scores are still within 25% of scores achieved by the extensive search. This justifies the need to select a proxy model capable of accurately predicting how well a corresponding tuned algorithm will perform on a given dataset.

Selecting hyperparameters for proxy models is a challenging task. For instance, Figure 13 compares the performance of running our pipeline with our proxy models versus models with scikit-learn default hyperparameters across our test corpus, with HyperGD disabled. The proxy models mostly outperform their default scikit-learn counterparts. In particular, using proxy models gives an average score ratio improvement of 2.75×, while keeping the average runtime almost the same.

Figure 14 shows the sensitivity of our proxy model scores to the number of datasets used to metalearn their hyperparameters. The y-axis shows the ratio of our proxy model score to respective default scikit-learn model score, averaged across all models and test corpus datasets. Our proxy models, on average, outperform the scikit-learn defaults. Furthermore, as expected, with more datasets, our proxy models improve. We see that even 75% of the train corpus (98 out of 130 datasets) is sufficient to metalearn our proxy models.

5.3 Algorithm Selection

We evaluate the accuracy of our algorithm selection approach in our pipeline by comparing it against an exhaustive algorithm selection scenario, where the pipeline tries all algorithms on a given dataset. We do this by specifying every algorithm in our pipeline, running the pipeline for every

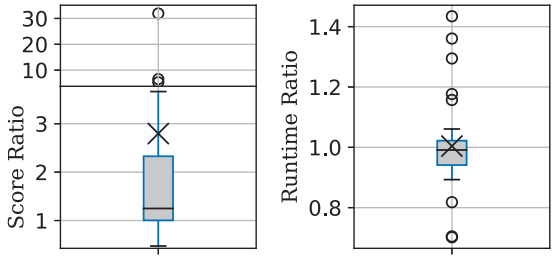


Figure 13: Oracle AutoML score and runtime with proxy models normalized to scikit-learn default models. HyperGD is disabled for this experiment. Score/runtime ratios >1 mean proxy models are better/faster than defaults.

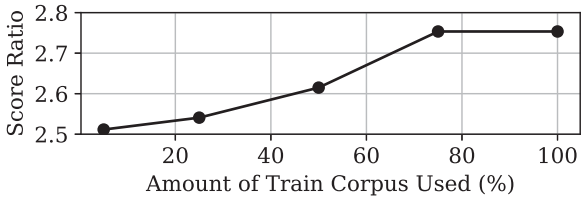


Figure 14: Sensitivity of proxy models to the number of datasets from train corpus used to select their hyperparameters. Score ratio is relative to default scikit-learn models. Ratio >1 means proxy models are better.

one of those algorithms, and picking the algorithm with the highest cross-validation score. We then compare the exhaustive algorithm selection test scores and runtimes against the Oracle AutoML experiments with algorithm selection enabled. Figure 15 shows the score and runtime for the Oracle AutoML normalized to the exhaustive algorithm selection runs across all test datasets. We see that enabling algorithm selection in our pipeline results in an average score loss of just 4.77% while giving a runtime advantage of $\sim 4.5\times$ over exhaustive algorithm selection.

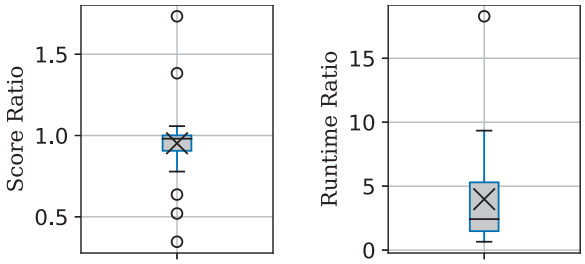


Figure 15: Evaluation of algorithm selection. The y-axis shows the score achieved by Oracle AutoML normalized to the scores achieved by exhaustive algorithm selection. Score/runtime ratios >1 mean Oracle AutoML is better/faster than exhaustive algorithm selection.

5.4 Adaptive Data Reduction

The main goal of ADR is to provide speedup for subsequent pipeline stages with minimal test score loss. To demonstrate ADR in action, we show the details of score versus sample size (Figure 16-top) and feature length (Figure 16-bottom) for one dataset (1503). ADR selects $\sim 30k$ rows out of the total of $\sim 148k$ rows and 13 out of 14 features as a representative sample of the full dataset. This results in $\sim 4\times$ pipeline speedup for this dataset with negligible score loss.

To further show the benefits of ADR, we disable it in our pipeline and measure the relative score loss over the full pipeline runs across our test datasets (Figure 17). Enabling ADR reduces our pipeline's average runtime by more than 8.73%, while providing an average score improvement of 1.80%. As expected, this score improvement and speedup are much more pronounced for large datasets from Table 2, with 35.98% speedup and 3.65% score improvement.

5.5 HyperGD

HyperGD design aims to achieve fast hyperparameter tuning without compromising on model performance. We evaluate HyperGD on its ability to tune our proxy models for the respective algorithm chosen by algorithm selection on the data sample returned by ADR. Figure 18 shows the

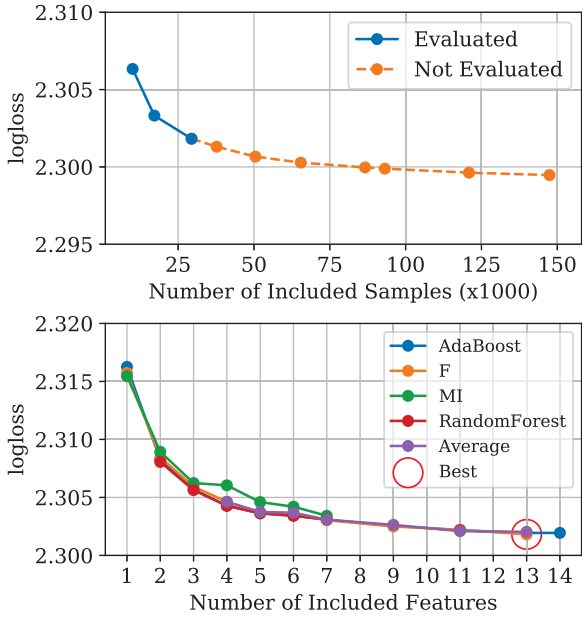


Figure 16: ADR on dataset 1503. Row sampling illustrates cross-validation score convergence on a sample size of $\sim 30k$ samples. Feature selection settles on 13 out of 14 features.

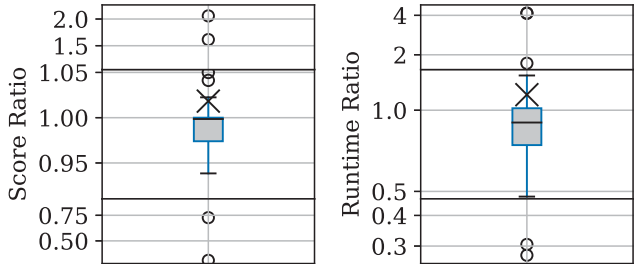


Figure 17: Evaluation of ADR. Score and runtime achieved by Oracle AutoML with ADR enabled, normalized to Oracle AutoML with ADR disabled. Ratios >1 mean Oracle AutoML with ADR enabled is better/faster.

benefits of HyperGD in Oracle AutoML. The plots on the left and right, respectively, show the improvements in cross-validation and test scores (across test corpus) gained by running HyperGD. As expected, HyperGD always improves the cross-validation score, given that the proxy model hyperparameters are a part of the bootstrap default set. The test score improves by 5.8% on average over the proxy models. There is only one dataset (53) for which the test score degrades by $\sim 32\%$ over the proxy model. Dataset 53 is a very small dataset (216 rows) and this level of generalization error is possible even with five-fold cross-validation.

To demonstrate the highly-parallel and asynchronous nature of HyperGD, we perform a detailed analysis of one dataset in the test corpus. Figure 19 shows the timeline for HyperGD for dataset 351 running in both *synchronous* (implemented for this experiment) and *asynchronous* modes. Our asynchronous optimization improves HyperGD runtime by more than 34%, while obtaining a similar score as the synchronous optimizer. As it can be seen from Figure 19, synchronization between batches (best values of individual hyperparameters) is done much more frequently in the asyn-

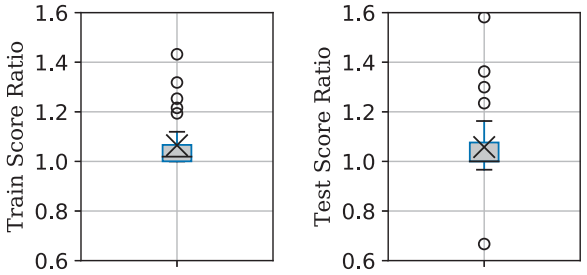


Figure 18: Oracle AutoML train and test scores with HyperGD enabled normalized to HyperGD disabled. Score ratios >1 mean HyperGD improves the score.

chronous version. This eliminates long tails due to stragglers, resulting in speedup compared to the synchronous version. In the figure, in synchronous mode, the second batch of trials is not queued until all the results from the first batch are evaluated. This results in a delay of ~ 100 seconds to start queuing the second batch of trials. This delay can worsen if there is at least one straggler in a batch, which can be seen in the last three batches of Figure 19(left), where a single straggler more than triples the batch's runtime. Conversely, in the asynchronous version, the second batch delay is only ~ 30 seconds, as we only wait for a minimum number of evaluations from the current batch to complete before queuing the next batch.

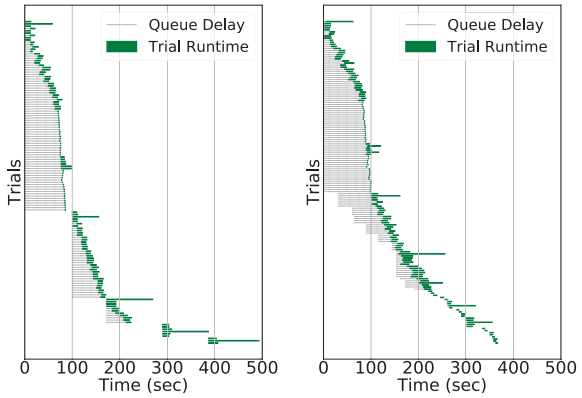


Figure 19: Synchronous (left) versus asynchronous (right) hyperparameter optimization. Different batches of trials showing queue and completion times.

6. LESSONS LEARNED

In the process of research and development of Oracle AutoML, we made a few important observations that are often overlooked but are critical to making an AutoML tool practical. Some, like the need for mitigating stragglers throughout the pipeline, are apparent from our design. In this section, we discuss some less known, subtle issues that we faced repeatedly during this research in an industrial setting:

Continual optimization versus convergence. Practical work cycle of a data scientist or ML application developer often involves repeatedly iterating over a model-build-test-repeat cycle with a human in the loop. Optimizing for the last mile during each iteration is often unnecessary. The onus is on the tool to figure out when to stop (convergence) and make the data analyst more productive.

Inter- versus intra-model parallelism. Utilizing the available cores or nodes efficiently is critical, especially in a pay-per-use cloud compute setting. Parallelizing multiple model evaluations (inter-) across all available compute is often reasonable but may not be optimal for all ML algorithms. For instance, linear algorithms, like LogisticRegression, may not benefit much from intra-model parallelism. However, an XGBoost model achieves $\sim 4\times$ improvement when balancing available compute between inter- and intra-model parallelism. Thus, achieving a good balance in parallelism can have a significant impact on overall AutoML speed.

Repeatability is challenging. Given the large search space in the AutoML problem, highly parallelized random search can result in acceptable optimization outcome. However, from a practical standpoint in an industrial setting, repeatability of AutoML output given the same set of inputs is often critical for global compliance and explainability. Hence, optimizing the pipeline without relying on randomness, even when theoretically necessary (e.g., random perturbation to get out of a local maxima), is challenging.

Metalearning is tedious but rewarding. Offline learning on wide-variety of datasets and algorithms can be very rewarding as demonstrated by Oracle AutoML. However, this comes at the cost of repeating tedious experiments when there is any significant change to the underlying system, like changes to ML algorithm implementation, or platform upgrades (e.g., Python 2 to 3). Generalizing learning beyond such disruptive changes is an open research problem.

7. CONCLUSION

We have shown that model performance does not need to be compromised to achieve speed in an AutoML pipeline. We presented novel techniques for algorithm selection, adaptive data reduction, and hyperparameter tuning. By using metalearning in a targeted manner to define our proxy models, we showed Oracle AutoML is able to make accurate algorithm and data reduction choices, enabling iteration-free optimization. This, in turn, resulted in an efficient pipeline.

We are working on several exciting frontiers to further improve our pipeline. In particular, we are working on making Oracle AutoML cloud-native, to further take advantage of the Oracle Cloud. The latter entails enhanced runtime cost models, further leveraging our proxy model approach. We are also working on more complex metalearned models for domain specific tasks to improve fault tolerance and provide additional visibility into the AutoML process to consumers, thereby enhancing user experience.

8. REFERENCES

[1] Amazon. Sagemaker. `https://aws.amazon.com/blogs/aws/amazon-sagemaker-autopilot-fully-managed-automatic-machine-learning/`, 2019.

[2] K. Anumalasetty. New automated machine learning capabilities in azure machine learning service. `https://azure.microsoft.com/en-us/blog/new-automated-machine-learning-capabilities-in-azure-machine-learning-service/`, 2018.

[3] A. Balaji and A. Allen. Benchmarking automatic machine learning frameworks. *arXiv preprint arXiv:1808.06492*, 2018.

[4] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012.

[5] J. Bergstra, D. Yamins, and D. Cox. Hyperopt: a python library for optimizing the hyperparameters of machine learning algorithms. *Proc. SciPy 2013*, page 13, 2013.

[6] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–115–I–123. JMLR.org, 2013.

[7] G. Brown, A. Pocock, M.-J. Zhao, and M. Luján. Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. *J. Mach. Learn. Res.*, 13(1):27–66, Jan. 2012.

[8] N. V. Chawla. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 875–886. Springer, 2009.

[9] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[10] W. G. Cochran. Sampling techniques-3. 1977.

[11] S. Ertekin, J. Huang, L. Bottou, and L. Giles. Learning on the border: active learning in imbalanced data classification. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 127–136. ACM, 2007.

[12] S. Falkner, A. Klein, and F. Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446, 2018.

[13] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 2755–2763, Cambridge, MA, USA, 2015. MIT Press.

[14] M. Feurer, J. T. Springenberg, and F. Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 1128–1135. AAAI Press, 2015.

[15] Y. Fogel and M. Feder. On the problem of on-line learning with log-loss. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 2995–2999. IEEE, 2017.

[16] N. Fusi, R. Sheth, and M. Elibol. Probabilistic matrix factorization for automated machine learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3348–3357. Curran Associates, Inc., 2018.

[17] P. Gijsbers, E. LeDell, J. Thomas, S. Poirier, B. Bischl, and J. Vanschoren. An open source automl benchmark. *arXiv preprint arXiv:1907.00909*, 2019.

[18] T. A. F. Gomes, R. B. C. Prudêncio, C. Soares, A. L. D. Rossi, and A. Carvalho. Combining meta-learning and search techniques to select parameters for support vector machines. *Neurocomput.*, 75(1):3–13, Jan. 2012.

[19] J. González, Z. Dai, P. Hennig, and N. D. Lawrence. Batch Bayesian Optimization via Local Penalization. *ArXiv e-prints*, May 2015.

[20] P. Hall, M. Kurka, and A. Bartz. *Using H2O Driverless AI*, January 2018.

[21] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering*, (9):1263–1284, 2008.

[22] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536, 2017.

[24] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[25] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *J. Mach. Learn. Res.*, 18(1):826–830, Jan. 2017.

[26] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu. Feature selection: A data perspective. *ACM Comput. Surv.*, 50(6):94:1–94:45, Dec. 2017.

[27] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, Jan. 2017.

[28] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar. A system for massively parallel hyperparameter tuning. *arXiv:1810.05934v5*, 2018.

[29] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang. Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. *PVLDB*, 11(5):607–620, 2018.

[30] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[31] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1710.06952*, 2017.

[32] Lightgbm. Lightgbm v2.2.3 python package documentation. `https://github.com/microsoft/LightGBM/blob/v2.2.3/docs/Features.rst`, 2018.

[33] J. Liu, S. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *International Conference on Machine Learning*, pages 469–477, 2014.

[34] G. Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):18, 2016.

[35] matplotlib. Auto-sklearn git repository.

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html, 2019.

[36] L. C. Molina, L. Belanche, and A. Nebot. Feature selection algorithms: a survey and experimental evaluation. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 306–313, 2002.

[37] M. Msr and M. Sebag. Alors: An algorithm recommender system. *Artificial Intelligence*, 244, 12 2016.

[38] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[39] Oracle, Inc. The Oracle AutoML Pipeline. https://docs.cloud.oracle.com/en-us/iaas/tools/ads-sdk/1.0.0/user_guide/automl/overview.html, 2020.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[41] Z. Peng, Y. Xu, M. Yan, and W. Yin. Arock: an algorithmic framework for asynchronous parallel coordinate updates. *SIAM Journal on Scientific Computing*, 38(5):A2851–A2879, 2016.

[42] V. Perrone, R. Jenatton, M. W. Seeger, and C. Archambeau. Scalable hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 6845–6855, 2018.

[43] B. Pfahringer, H. Bensusan, and C. G. Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 743–750, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[44] P. Poski. Automl comparison. https://mljar.com/blog/automl-comparison/, 2017.

[45] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[46] M. Reif, F. Shafait, and A. Dengel. Meta-learning for evolutionary parameter optimization of classifiers. *Machine Learning*, 87(3):357–380, Jun 2012.

[47] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The evolution of data to life-critical dont focus on big data. *Focus on the Data Thats Big Sponsored by Seagate The Evolution of Data to Life-Critical Dont Focus on Big Data*, 2017.

[48] J. R. Rice et al. The algorithm selection problem. *Advances in computers*, 15(65-118):5, 1976.

[49] scikit-learn. Definition of sklearn metrics log_loss score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html, 2019.

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

[51] sklearn. Sklearn v0.20 home page. https://scikit-learn.org/0.20/, 2018.

[52] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012. Curran Associates Inc.

[53] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, P. Prabhat, and R. P. Adams. Scalable bayesian optimization using deep neural networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2171–2180. JMLR.org, 2015.

[54] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 368–380, New York, NY, USA, 2015. ACM.

[55] Y. Sun, M. S. Kamel, A. K. Wong, and Y. Wang. Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognition*, 40(12):3358–3378, 2007.

[56] L. Sun-Hosoya, I. Guyon, and M. Sebag. Activmetal: Algorithm recommendation with active meta learning. In *IAL@PKDD/ECML*, 2018.

[57] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. 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 '13, pages 847–855, New York, NY, USA, 2013. ACM.

[58] I. Tomek. Two modifications of cnn. *IEEE Trans. Systems, Man and Cybernetics*, 6:769–772, 1976.

[59] A. Truong, A. Walters, J. Goodsitt, K. Hines, B. Bruss, and R. Farivar. Towards automated machine learning: Evaluation and comparison of automl approaches and tools. *arXiv preprint arXiv:1908.05557*, 2019.

[60] A. Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[61] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[62] V. Vovk. The fundamental nature of the log loss function. In *Fields of Logic and Computation II*, pages 307–318. Springer, 2015.

[63] C. Wong, N. Houlsby, Y. Lu, and A. Gesmundo. Transfer learning with neural automl. In *Advances in Neural Information Processing Systems*, pages 8356–8365, 2018.

[64] S. J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[65] XGBoost. Xgboost v0.18 python package documentation. https://xgboost.readthedocs.io/en/release_0.81/, 2018.

[66] Y. Xu, C. Sutcher-Shepard, Y. Xu, and J. Chen. Asynchronous parallel adaptive stochastic gradient methods. *arXiv preprint arXiv:2002.09095*, 2020.

[67] D. Yogatama and G. Mann. Efficient Transfer Learning

Method for Automatic Hyperparameter Tuning. In S. Kaski and J. Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 1077–1085, Reykjavik, Iceland, 22–25 Apr 2014. PMLR.