Bandit Limited Discrepancy Search and Application to Machine Learning Pipeline Optimization

Akihiro Kishimoto[1], Djallel Bouneffouf[1], Radu Marinescu[1], Parikshit Ram[1], Ambrish Rawat[1], Martin Wistuba[2*], Paulito Palmes[1], Adi Botea[3*]

[1] IBM Research
[2] Amazon Research
[3] Eaton

Akihiro.Kishimoto@ibm.com, Djallel.Bouneffouf@ibm.com, radu.marinescu@ie.ibm.com, parikshit.ram@ibm.com, Ambrish.Rawat@ie.ibm.com, marwistu@amazon.com, paulpalmes@ie.ibm.com, adi.botea@eaton.com

Abstract

Optimizing a machine learning (ML) pipeline has been an important topic of AI and ML. Despite recent progress, pipeline optimization remains a challenging problem, due to potentially many combinations to consider as well as slow training and validation. We present the BLDS algorithm for optimized algorithm selection in a fixed ML pipeline structure. BLDS performs multi-fidelity optimization for selecting ML algorithms trained with smaller computational overhead, while controlling its pipeline search based on multi-armed bandit and limited discrepancy search. Our experiments on classification benchmarks show that BLDS is superior to competing algorithms. We also combine BLDS with hyperparameter optimization, empirically showing the advantage of BLDS.

Introduction

Automated Machine Learning (AutoML) seeks to automatically compose and parameterize ML algorithms to maximize a given metric such as predictive accuracy on a given dataset. The task has received increased attention over the past decades especially in light of the recent explosion in ML applications. AutoML has gradually extended from hyperparameter optimization (HPO) for the best configuration of a single ML algorithm (Bergstra et al. 2011) to tackling the optimization of the entire ML pipeline from data preparation to model learning (Feurer et al. 2015a). This effort has spurred the development of a wide variety of efficient AutoML systems, e.g., (Kotthoff et al. 2017; Olson et al. 2016; Feurer et al. 2015a; Mohr, Wever, and Hullermeier 2018; Rakotoarison, Schoenauer, and Sebag 2019; Liu et al. 2020).

A typical ML pipeline often consists of a fixed sequence of successive stages such as pre-processing, feature selection, transformation and estimation. The AutoML problem known as the *combined algorithm selection and hyperparameter optimization (CASH)* selects the ML algorithm for each stage and the hyperparameters of these algorithms such that a given black-box objective function is optimized. The fixed structure of the pipeline implies an optimization problem with a fixed number of decision variables where, for example, we have one variable for a preprocessing algorithm, one variable for a learning algorithm, and one variable for each parameter of each algorithm. This in turn leads to a complex solution space involving both discrete and continuous variables.

It has been shown recently that CASH is solved more efficiently by splitting the algorithm selection and the HPO into two simpler subproblems which are subsequently solved separately (Mohr, Wever, and Hullermeier 2018; Rakotoarison, Schoenauer, and Sebag 2019; Liu et al. 2020). However, the algorithm selection problem raises two challenges: (1) the black-box nature of the objective function prevents the algorithm selection from leveraging any of the objective function's characteristics in search for a better pipeline configuration; (2) there are many possible combinations of algorithms in a multi-stage pipeline structure and training every single pipeline configuration is very expensive, especially when dealing with large input datasets. In practice, this overhead is a major bottleneck because each evaluation of the black-box objective function to determine the value of the current pipeline configuration involves the *entire* training data.

We focus on the *algorithm selection* problem in AutoML and introduce Bandit Limited Discrepancy Search (BLDS), which combines ideas behind algorithms for the multi-armed bandit (MAB) problem, e.g., (Auer, Cesa-Bianchi, and Fischer 2002), limited discrepancy search (LDS) (Harvey and Ginsberg 1995) and multi-fidelity optimization (Sabharwal, Samulowitz, and Tesauro 2016).

Specifically, BLDS assumes that a better solution tends to exist in a set of pipelines similar to the current best one and uses the notion of discrepancy to reduce the search space examined. However, a series of BLDS' local search may lead to an optimized pipeline that is very *different* from the initial one. In addition, we design BLDS to reduce the computational overhead associated with training the pipelines on large datasets. Our BLDS algorithm is the first multi-fidelity optimization method designed specifically to address the algorithm selection subproblem. The algorithm starts with a small subset of training data and it gradually increases this subset if the corresponding pipeline is promising. BLDS involves a procedure to compare pipelines trained with different subsets of data. Unlike existing multi-fidelity optimization approaches such as the Data Allocation Upper Bound (DAUB) algorithm (Sabharwal, Samulowitz, and Tesauro 2016), BLDS calculates the upper and lower confidence

*Work performed while at IBM Research.

bounds which are inspired by MAB but which additionally account for multi-fidelity optimization.

We show a theoretical property on the confidence bounds which allow BLDS to select a more promising pipeline as well as to decide whether to assign more resources to a pipeline for further training. Besides, BLDS is naturally incorporated into any scheme that splits CASH into two separate subproblems. We combine BLDS with HPO under the recent ADMM framework of Liu et al. (2020), thus developing a scheme to *warm-start* a BLDS run with previous runs.

We compare BLDS with the Combinatorial MAB (CMAB) algorithm (Liu et al. 2020) as well as DAUB and Hyperband (Li et al. 2018), which we adapt for algorithm selection. Using well-known benchmarks and a fixed 4-stage pipeline structure comprising over 3000 possible algorithm selections, we show that BLDS performs better than the competing methods. We also empirically show that BLDS yields better objective values than the competitors when it is combined with HPO, demonstrating that our approach can solve CASH more effectively on large datasets.

Related Work

Existing AutoML systems can be divided into two categories: methods that use the entire training dataset during optimization and multi-fidelity optimization methods that start with a small fragment of training data and gradually increase it as the optimization progresses.

Systems from the first category typically design the CASH problem as an optimization problem over a high-dimensional joint search space with discrete and continuous variables, which is then solved in many different ways, including Bayesian optimization (BO) (Bergstra et al. 2011), mixed integer linear programming (Liu et al. 2020), evolutionary algorithms (Olson et al. 2016), AI planning (Mohr, Wever, and Hullermeier 2018; Katz et al. 2020), Monte Carlo tree search (MCTS) coupled with BO (Rakotoarison, Schoenauer, and Sebag 2019). More specifically, Auto-WEKA (Kotthoff et al. 2017) and Auto-sklearn (Feurer et al. 2015a) apply the general purpose algorithm configuration framework SMBO (Hutter, Hoos, and Leyton-Brown 2011) based on BO to find optimal ML pipelines. TPOT (Olson et al. 2016) and its extensions like RECIPE (de Sa et al. 2017), Auto-MEKA (de Sa, Freitas, and Pappa 2018), Auto-Stacker (Chen et al. 2018) or Auto-DSGE (Assuncao et al. 2020) discretize the continuous hyperparameters and use genetic programming together with a context-free grammar to evolve randomly generated pipelines. AlphaD3M (Drori et al. 2018) applies reinforcement learning for solving CASH.

None of the methods above impose an efficient decomposition over hyperparameters and algorithm selection. ML-Plan (Mohr, Wever, and Hullermeier 2018) is the first system that distinguishes explicitly between the algorithm selection and HPO subproblems and uses *hierarchical task networks* based planning to solve them efficiently. ReinBo (Sun, Lin, and Bischl 2020) employs reinforcement learning for algorithm selection and BO for HPO. MOSAIC (Rakotoarison, Schoenauer, and Sebag 2019) partially splits CASH into algorithm selection and HPO, utilizing MCTS and BO respectively for each of the subproblems, which are coupled with a shared surrogate model. The more recent ADMM system splits the algorithm selection phase and HPO into two simpler subproblems which are subsequently solved separately in an iterative manner using the augmented Lagrangian function and the alternating direction method of multipliers (ADMM) (Liu et al. 2020). In practice, CMAB for algorithm selection and BO for HPO are shown to perform best under the ADMM framework (Liu et al. 2020).

Multi-fidelity optimization using a small subset of the original training data to approximate the loss function has been considered in the context of continuous search spaces and HPO, e.g., (Klein et al. 2017; Falkner, Klein, and Hutter 2018; Lu et al. 2019). These approaches could be used to address CASH (e.g., run them under the SMBO framework), but how to fit them well into CASH still remains an open question. Successive halving (Karnin, Koren, and Somekh 2013; Jamieson and Talwalkar 2016) and Hyperband are shown to be competitive for HPO. DAUB applies multi-fidelity optimization to effectively train a fixed set of classifiers.

Approaches for empirically predicting the *learning curve* also increase the training data size (Figueroa et al. 2012; Koshute, Zook, and McCulloh 2021; Mohr and van Rijn 2021). Under the assumption that using the entire training dataset does not necessarily yield the best accuracy, they attempt to predict the case where a model cannot be improved with additional training examples. In contrast, multi-fidelity optimization aims at training the most promising model with the entire dataset and removing unpromising ones with their approximations calculated by the subsets of the dataset.

Other AutoML tasks include pipeline generations, e.g., PIPER (Marinescu et al. 2021) and TPOT. However, the search space of pipeline generation is different from that of pipeline optimization with a fixed structure.

Preliminaries

Algorithm Selection An ML pipeline structure consists of a fixed sequence of m stages (e.g., data preprocessor $\rightarrow$ feature preprocessor $\rightarrow$ classifier) such that for each stage $j = 1, \ldots, m$ there is a set $\mathcal{A}_j$ of available ML algorithms. A pipeline configuration (or pipeline for short) $p \in \mathcal{P}$ is a complete configuration of algorithms, one for each stage, namely $p = (a_1, \ldots, a_m)$ where $a_j \in \mathcal{A}_j$ is the algorithm selected for the j-th stage and $\mathcal{P} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$ is the set of all possible pipeline configurations. Given a limited amount of training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, the goal of the algorithm selection problem is to determine the pipeline $p^* \in \mathcal{P}$ with optimal generalization performance. Generalization performance is estimated by splitting $\mathcal{D}$ into disjoint training and validation sets T and V, learning a function f by applying p^* to T and evaluating the predictive performance of function f on V. Therefore, the algorithm selection problem is written as $p^* = \arg\min_{p \in \mathcal{P}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(p, T^{(i)}, V^{(i)})$, where $\mathcal{L}(p, T^{(i)}, V^{(i)})$ is the value of the black-box loss function (e.g., classification error) achieved by p when trained on $T^{(i)}$ and evaluated on $V^{(i)}$. We use k-fold cross-validation (Kohavi 1995), which splits the training data into k equal-sized partitions $V^{(1)}, \ldots, V^{(k)}$, and sets $T^{(i)} = \mathcal{D} \setminus V^{(i)}$ for

Algorithm 1: LDS: Limited Discrepancy Search

```
1: procedure LDS
2:   for all k = 0 ... n do
3:     if PROBE(root, k) then return true
4: function PROBE(node, k)
5:   if isLeaf(node) then return isGoal(node)
6:   if k = 0 then return PROBE(left(node), 0)
7:   else return PROBE(right(node), k-1) or PROBE(left(node),k)
```

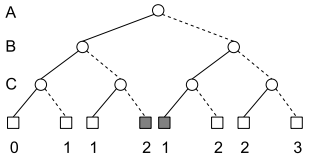


Figure 1: Search space traversed by LDS (the height is 3). The number of discrepancies is indicated below the leaf nodes.

$i = 1, \ldots, k$.

Limited Discrepancy Search We consider a search space that is a *complete binary tree* with bounded height h. Leaf nodes correspond to *goals* or *failures* and the task of interest is to find a goal. Each internal node represents a decision that must be made to reach a goal. Furthermore, the left child of each internal node represents following the recommendation of a value-ordering heuristic and the right child represents going against that recommendation. Disregarding the heuristic recommendation is called a *discrepancy*. The number of discrepancies of a leaf node is the number of right turns in the path from the root to that leaf node. *Limited Discrepancy Search* (LDS) (Harvey and Ginsberg 1995; Korf 1996) is a depth-first search algorithm that searches for a goal while iteratively increasing the number of discrepancies. The pseudo-code is given in Algorithm 1. The k-th iteration of the main loop visits all the leaves having k or fewer discrepancies. Function PROBE is a standard recursive implementation of depth-first search such that: (i) it keeps track (parameter k) of the number of discrepancies still available, (ii) if a discrepancy is consumed, k is decreased before the recursive call and (iii) if no further discrepancies are available, LDS does not disregard the heuristic. Since the last iteration visits all the leaves, LDS is complete. In practice, LDS is used in a anytime manner until a solution is found or a time limit is reached. In our case, each layer in the search tree stands for a stage in the ML pipeline structure. An edge represents an instantiation of one algorithm in that stage and the terminal nodes correspond to pipeline configurations. We also use the notion of discrepancy without any value-ordering heuristic.

Example 1. *Figure 1 shows a search tree with height 3. The gray leaves correspond to goals (solutions). LDS stops during iteration $k = 1$ where it finds the solution with 1 discrepancy.*

Bandit Limited Discrepancy Search

We present our new *Bandit Limited Discrepancy Search* (BLDS) algorithm for tackling the algorithm selection problem in AutoML. The basic idea behind BLDS is to conduct a discrepancy-based exploration that focuses on the most promising portions of the pipeline search space while controlling the size of the training data used for training the pipelines found in a most cost-effective manner. Unlike standard LDS, we consider an optimization problem where each leaf node is a goal and corresponds to a pipeline p which has an associated cost (i.e., value of a black-box loss or objective function $\mathcal{L}(p)$) and the task is to find the least-cost one.

Algorithm 2: BLDS: Bandit Limited Discrepancy Search

```
Require: Training set T, validation set V, discrepancy disc
 1: procedure BLDS(T, V, disc)
 2:   while time is not up do
 3:     p_init = FINDINITIALPIPELINE()
 4:     repeat
 5:       INCREASEANDTRAIN(p_init, T, V)
 6:       for all (θ = 1; θ ≤ disc; θ = θ + 1) do
 7:         p_new = SEARCH(p_init, p_init.lcb, p_init.ucb, T, V, 1, θ)
 8:         if (p_new ≠ φ) then
 9:           p_init = p_new; break
10:     until (p_init is trained with a full set of T)
11:   return best pipeline obtained
12: function SEARCH(Pipeline p, LCB lcb, UCB ucb, training set
    T, validation set V, stage i, current discrepancy θ)
13:   if (θ = 0 ∨ i > m) then
14:     (l, u) = GETCURRENTPERFORMANCE(p)
15:     if (u < lcb) then return p
16:     if (l ≤ ucb) then (l, u) = INCREASEANDTRAIN(p, T, V)
17:     if (u < ucb) then return p
18:     else return φ
19:   else
20:     for all algorithm a ∈ A_i do
21:       if (p[i] = a) then
22:         r = SEARCH(p, lcb, ucb, T, V, i + 1, θ)
23:       else
24:         p_new = p; p_new[i] = a
25:         r = SEARCH(p_new, lcb, ucb, T, V, i + 1, θ − 1)
26:       if (r ≠ φ) then return r
27:     return φ
```

Algorithm Description

Algorithm 2 describes the BLDS algorithm. We consider a linear[1] pipeline structure with m stages such that each stage i has a set $\mathcal{A}_i$ of available ML algorithms. In the pseudo code, function FINDINITIALPIPELINE generates a randomly initialized pipeline $p_{init} = (a_1, \ldots, a_m)$, however, BLDS can start with any pipeline obtained with other AutoML methods. Function INCREASEANDTRAIN trains pipeline p with a (sub)set of training data T and evaluates p's performance on validation data V. As also discussed in (Sabharwal, Samulowitz, and Tesauro 2016), when training p for the k-th time, BLDS selects $b\eta^k$ samples from T, where b and η are constants. The pipeline p has an extra structure to preserve an objective value v and two associated values lcb and ucb which refer to a lower confidence bound (LCB) and an upper confidence bound (UCB), respectively. The LCB and UCB values

[1]BLDS is applicable to non-linear pipeline structures representing trees/DAGs as well.

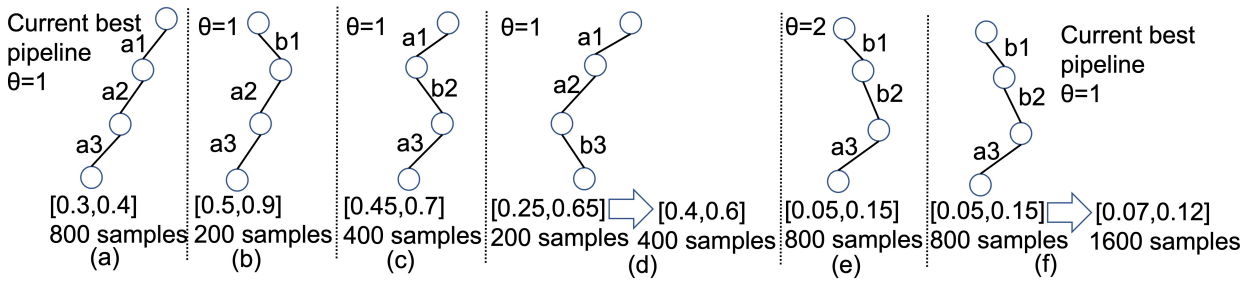


Figure 2: Search behavior of BLDS ($\eta = 2$) for a three-step pipeline structure where each module has two algorithm choices

are regarded as lower and upperbounds of the achievable objective value (see the next subsection for details). The discrepancy $disc$ indicates the maximum number of allowed algorithm changes to the stages of the initial pipeline p_{init}. BLDS assumes that a better pipeline tends to be instantiated in a similar fashion to p_{init} and, therefore, it examines a limited search space where similar pipelines are located. The symbol ϕ is used to indicate that the algorithm could not find a pipeline better than p_{init} with current discrepancy value θ.

The algorithm starts with a discrepancy value θ of 1 and conducts an iterative search that allows to change the algorithms of at most θ stages in p_{init} while incrementing θ until a better pipeline p_{new} is found or θ exceeds $disc$ (see lines 5-10). If p_{new} is found, BLDS uses it as a new initial pipeline and attempts to improve it further. BLDS repeats the steps of IncreaseAndTrain and the iterative search limited by θ until it finds a pipeline trained with a full set T. However, even after finding such a pipeline, the algorithm can still continue the search for another one by restarting with a different initial pipeline calculated by FindInitialPipeline until it uses up the allocated time. For efficiency, BLDS also caches the objective value and the corresponding UCB/LCB values for all trained pipelines in order to avoid retraining them.

Function Search (lines 12-27) performs the actual exploration of the pipeline search space limited by discrepancy θ. Specifically, when it selects an algorithm a that is different from the one corresponding to stage i in p_{init}, it decrements θ to reduce the number of changes allowed for the remaining stages (lines 24-25). Otherwise, the algorithm for stage i remains unchanged and, therefore, the θ value is preserved (line 21-22). When Search either has checked all m stages in p or consumed the discrepancy budget, it checks p's performance (lines 13-18). The GetCurrentPerformance method retrieves p's LCB and UCB values l and u if they are cached. Caching alleviates the overhead of revisiting the same pipelines possibly with different θ. Otherwise, it evaluates p with V after training p with b samples in T. BLDS assumes the real objective value for p to be in $[l, u]$ and decides whether or not p is a promising pipeline as well as whether or not p should be trained with a larger training subset. This way, BLDS attempts to focus on promising pipelines and alleviates the training overhead. Search receives p_{init}'s LCB and UCB values lcb and ucb. If $u < lcb$ holds, p is considered to be better than p_{init} and becomes a new pipeline to start with (line 15). If $l \leq ucb$ (and $lcb \leq u$) holds, p might or might not be better than p_{init}. In this case, p is retrained with an increased training (sub)set and re-evaluated with V. The algorithm subsequently selects a pipeline based on whether or not p's updated UCB value is better than that of p_{init} (lines 16-17). For the other cases (e.g., $ucb \leq l$ holds), p_{init} is considered to be better than p and is kept (line 18).

Example 2. *Figure 2 illustrates the execution of BLDS on a 3-stage pipeline structure. The UCB and LCB values u and l of a pipeline are written as $[l, u]$. Let $p_1 = (a1, a2, a3)$ be the current best pipeline trained with 800 samples (Fig. 2(a)). BLDS limits search with $\theta = 1$, allowing to change only one stage in p_1. Figures 2(b)-(d) show the pipelines examined with $\theta = 1$. In Figure 2(b), BLDS examines pipeline $p_2 = (b1, a2, a3)$ but considers that p_1 is better than p_2, since p_2's LCB value is larger than p_1's UCB value. So is the case for pipeline $(a1, b2, a3)$ (Fig. 2(c)). In Figure 2(d), there is an overlap between the regions of the LCB and UCB values for pipelines p_1 and $p_3 = (a1, a2, b3)$. To obtain a more accurate objective value, BLDS re-trains p_3 with an increased training subset (i.e., 400 samples) and updates its LCB and UCB values. BLDS finds that p_1 is still better than p_3. Since BLDS cannot find a pipeline better than p_1 with $\theta = 1$, it sets $\theta = 2$, allowing to modify any of two modules in p_1. BLDS reaches pipeline $p_4 = (b1, b2, a3)$ (Fig. 2(e)). The UCB value of p_4 is smaller than p_1's LCB value, indicating that p_4 is better than p_1. Therefore, BLDS stops searching with $\theta = 2$, sets p_4 to the new best pipeline and resets $\theta = 1$. By using p_4 as a new initial pipeline, BLDS re-trains p_4 with an increased training subset (i.e., 1600 samples) and obtains new LCB and UCB values. BLDS performs search with $\theta = 1$, allowing only one change to p_4.*

Upper and Lower Confidence Bounds

Existing approaches to compute UCB values for addressing the MAB problems only account for the number of visits to each arm[2]/branch, e.g., (Auer, Cesa-Bianchi, and Fischer 2002). However, the computational overhead of training a pipeline configuration depends on the size of the training data. The confidence of the accuracy of the pipeline performance is also associated with the training data size. In other words,

[2]Unlike (Kocsis and Szepesvári 2006), an arm corresponds to a pipeline configuration at a terminal node of BLDS' search tree.

two visits to the same pipeline configuration with different training data sizes need to be handled in a different way.

We adapt (Even-Dar, Mannor, and Mansour 2006) to algorithm selection. Our UCB and LCB formulas account for a relation between the total size of the training (sub)set invested so far and the accuracy of the objective value: $UCB = v + \sqrt{\frac{\log \frac{cLD_k{}^2}{\delta}}{D_k}}$ and $LCB = v - \sqrt{\frac{\log \frac{cLD_k{}^2}{\delta}}{D_k}}$, where v is an objective value for the k-th evaluation with validation set V and $c\ (> 4)$ and δ are constants, $L = \Pi_{i=1}^{m} |\mathcal{A}_i|$ and $D_k = \sum_{j=1}^{k} b\eta^{(j-1)}$.

The second term of UCB/LCB determines whether to perform a so-called *exploration*, aiming at updating v that might be inaccurate due to a small number of training examples.

Let $\mathcal{L}^{\max}$ be the upperbound of the loss $\mathcal{L}(\cdot, T, V)$ that we seek to minimize. For any pipeline $p_i, 1 \leq i \leq L$, let $q_i := \mathcal{L}^{\max} - \mathbb{E}_{T,V}\mathcal{L}(p_i, T, V)$ be the expected reward upon training on the full training set T, with an expectation over the sampling of the training and validation sets T and V respectively from the true data distribution. Assuming that all pipelines are always considered and that objective values are improved with a larger number of training examples, Theorem 1 refers to the upperbound of the size of the training data needed to differentiate the best pipeline from the others.

Theorem 1. *Suppose there are L pipelines $p_j\ (1 \leq j \leq L)$ sorted in an descending order of q_j. Denote $\Delta_i = q_1 - q_i > 0$ for $i = 2, 3, ..., L$. Finally, let t_i be the maximum number of times* IncreaseAndTrain *is invoked on a suboptimal pipeline $p_i, i \neq 1$ to find that p_1 is better than p_i. Then $\sum_{i=2}^{L} t_i$ is $O(\sum_{i=2}^{L} \frac{ln\left(\frac{L}{\delta\Delta_i}\right)}{\Delta_i^2})$ with probability at least $1 - \delta$.*

Proof. The proof follows exactly the same step as Theorem 8 in (Even-Dar, Mannor, and Mansour 2006), which also uses the exploration factor $c > 4$ in (Audibert, Bubeck, and Munos 2010). Our main argument is that, for any value of D_k and for any pipeline selection i, the observed reward $\hat{Q}_i^t$ is within α_t of q_i. For any D_k and pipeline selection i we have $P(|\hat{Q}_i^t - q_i| \geq \alpha_t) \leq 2e^{-2\alpha_t^2 D_k} \leq \frac{2\delta}{cLD_k^2}$. By taking the constant $c > 4$ and from the union bound we have that with probability at least $1 - \delta/L$ for any D_k and any pipeline selection $i, |\hat{Q}_i^t - q_i| < \alpha_t$. Therefore, with probability $1 - \delta$, the best pipeline is never eliminated. Furthermore, since α_t goes to zero as D_k increases, every non-best pipeline is eventually eliminated. To eliminate a non-best pipeline p_i we need to reach a time t_i such that $\hat{\Delta}_{t_i} = \hat{Q}_1^{t_i} - \hat{Q}_i^{t_i} \leq 2\alpha_{t_i}$. The definition of α_t combined with the assumption that $|\hat{Q}_i^t - q_i| \geq \alpha_t$ yields $\Delta_i - 2\alpha_t = (q_1 - \alpha_t) - (q_i + \alpha_t) \geq \hat{Q}_1 - \hat{Q}_i \geq 2\alpha_t$. This holds with probability at least $1 - \frac{\delta}{L}$ for $t_i = o(\frac{ln(L/\delta\Delta_i)}{\Delta_i^2})$. □

Theorem 1 indicates that $O(L \log L)$ trainings are needed to select the best pipeline, when Search always runs with discrepancy $\theta = m$. On the other hand, suppose that $O(T)$ is required to train each pipeline with the full training data of size T. A straightforward approach then requires $O(LT)$. Since pipeline optimization typically satisfies $\log L \ll T$, our approach significantly reduces the training overhead in practice. For example, consider $L = 10^{10}$ for a very complicated 10-step pipeline structure with 10 algorithm choices in each module optimized with a training data of size 10,000. Even in this case, $\log L = \log 10^{10} = 23 \ll T = 10000$. In the literature such as (Liu et al. 2020; Rakotoarison, Schoenauer, and Sebag 2019), while T can be much larger than 10,000, L is at most 7000 (a pipeline structure with 3-4 steps).

BLDS examines only a subset of pipelines limited by discrepancy $\theta < m$ and the size of the training data is fixed. Understanding the theoretical properties of these cases remains an open question.

Combination with HPO

Algorithm selection with BLDS is easily combined with HPO under the recent ADMM framework by Liu et al. (2020). Let the number of iterations of BLDS be the number of FindInitialPipeline calls. After BLDS is run for a preset number of iterations with training set T and validation set V, the best pipeline p_{best} found by BLDS is passed to an HPO algorithm subsequently run with T and V for a preset number of iterations to find better hyperparameters H for p_{best}. For the next BLDS run, H is used as a default hyperparameter set for p_{best}. These two alternating steps are repeated until the time limit is exceeded. To control the iteration size we employ the adaptation scheme suggested in (Liu et al. 2020).

Our BLDS implementation reuses the cached results of trained pipelines across different BLDS runs. However, the result for p_{best} is removed from the cache. Its cached result needs to be updated, since p_{best} might yield different performance due to its different hyperparameters. Our HPO implementation employs a BO-based approach (Shahriari et al. 2016). It receives BLDS' p_{best} trained with full T and performs HPO with full set T.

Extensions to DAUB and Hyperband

We adapt DAUB and Hyperband to obtain competing multi-fidelity optimization based baselines for algorithm selection.

DAUB For the training and validation sets T and V, our DAUB implementation generates all possible pipelines for a given fixed m-stage pipeline structure, and performs the steps described in (Sabharwal, Samulowitz, and Tesauro 2016). In the bootstrapping procedure, DAUB trains each pipeline with b, $b\eta$ and $b\eta^2$ training examples. DAUB calculates the upperbounds of the respective accuracy based on its linear regression model. DAUB's priority queue Q orders all the pipelines based on these upperbounds. DAUB then dequeues one pipeline p from Q and trains it with $\min(|T|, \eta N)$ training examples where N is the number of examples allocated to train it last time. DAUB then updates p's upperbound with a newly calculated estimated accuracy and enqueues p back in Q. DAUB repeats these steps until a most promising pipeline is trained with a full training set. Even if DAUB returns a first pipeline p trained with full T, it can continue running to return a second pipeline which is trained with full T and which might perform better than p with respect to an objective value evaluated with respect to V. Until DAUB's priority queue becomes empty, DAUB can keep searching for a better pipeline

Step	Module
1	Binarizer, Normalizer, Quantile transformer, MinMax scaler, Standard scaler, Robust scaler, KBins discretizer (ordinal encoding), None
2	Sparse random projection (dense output), PCA, RBF sampler, Gaussian random projection, Factor analysis (SVD=randomized), Fast ICA, Truncated SVD (algorithm=randomized), None
3	Select percentile, Select Fpr, Select Fdr, Select FweFS, Variance threshold, None
4	Random Forest, Gaussian NB, KNeighbors, Quadratic discriminant analysis, Extra trees, AdaBoost (base estimator=decision tree, max depth=3), Decision tree, Logistic regression

Table 1: ML modules (None indicates selecting no module)

in this way. In a combination with HPO under the ADMM framework, we define the number of DAUB's iterations as the number of pipelines with full T returned by DAUB. The pipelines not selected by DAUB are enqueued back to its priority queue. The selected pipeline is enqueued after its hyperparameters are optimized by an HPO algorithm.

Hyperband Our Hyperband implementation generates pipelines by random sampling and optimizes them by the steps described in (Li et al. 2018), which we define as one iteration.[3] In case of searching for pipelines without HPO, this iteration is repeated until the time is up. We set the maximum amount of resource (called parameter R) to the training data size. This allows to perform an increase of the training data subset in a similar way to DAUB and BLDS, while controlling the size of selected pipelines based on η. It also employs a caching scheme similar to that of BLDS, which effectively reuses previously trained pipelines within its Hyperband runs and among different Hyperband runs with a combination of HPO under the ADMM framework (Liu et al. 2020).

Experimental Results

We implemented all algorithms in Python with scikit-learn (Pedregosa, Varoquaux, and Gramfort 2011) and performed the experiments on a cluster of Intel Xeon CPU E5-2667 processors at 3.3GHz. When running the algorithms, we use only one core to better track the objective value versus time as suggested in (Rakotoarison, Schoenauer, and Sebag 2019; Liu et al. 2020). We evaluate: (a) BLDS(1) and BLDS(2) with $disc = 1, 2$, respectively, (b) CMAB (Liu et al. 2020), (c) DAUB, (d) Hyperband and (e) simple random search (RND). We also perform a comparison against MLDS and LDS. MLDS(1) and MLDS(2) perform multi-fidelity optimization and LDS with $disc = 1, 2$ but without using our UCB/LCB formulas. LDS(1) and LDS(2) perform LDS with $disc = 1, 2$ but without multi-fidelity optimization. For algorithm selection, note that LDS(1) can be regarded as Naive AutoML of Mohr and Wever (2021).

[3]While successive halving can be another choice, Hyperband tends to perform better in our preliminary experiments.

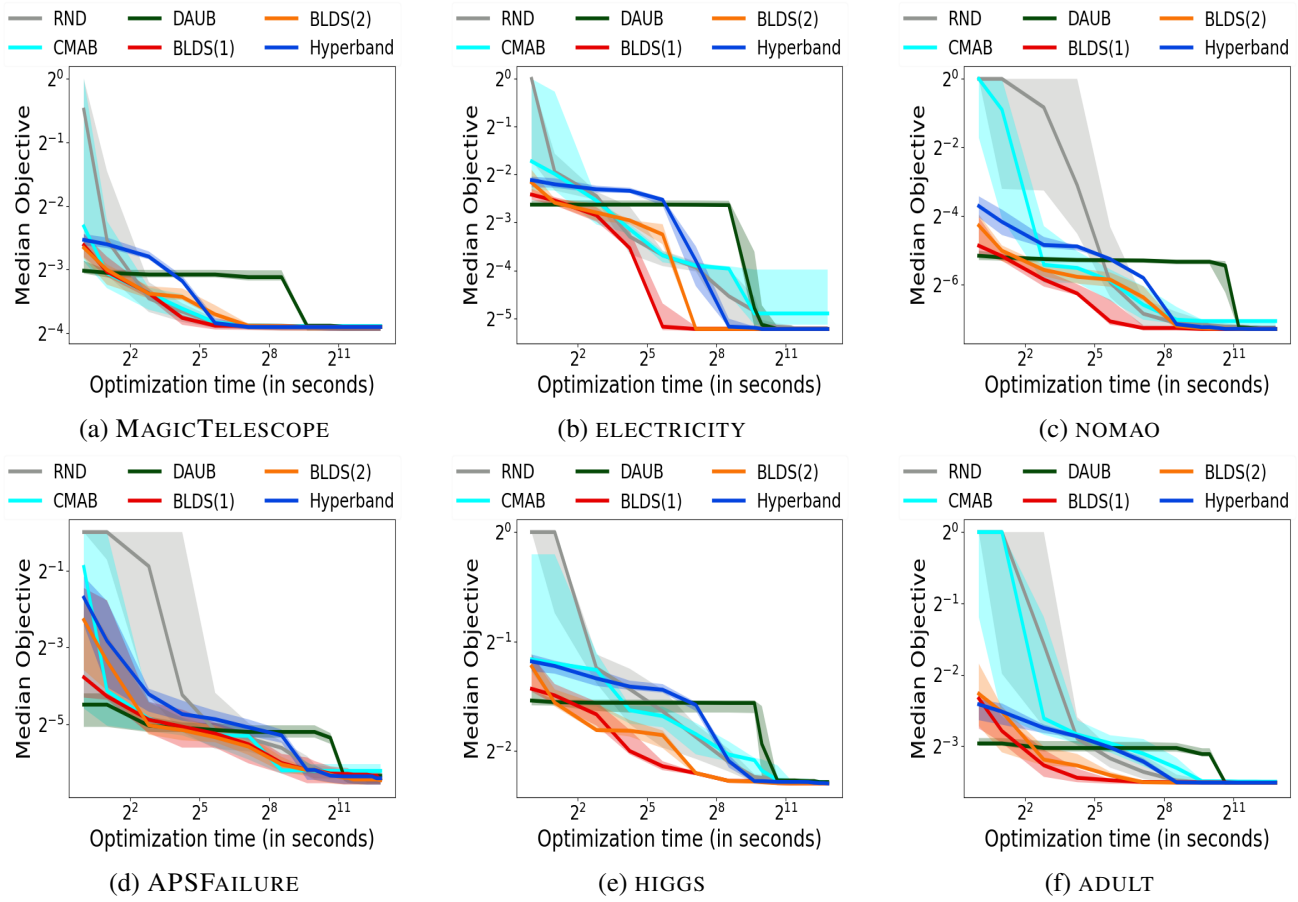
Setups We set up experiments similar to those in (Liu et al. 2020) and use standard evaluation metrics in the community (Feurer et al. 2015b). For each benchmark dataset, we consider a 70-30% train-validation split, and run each algorithm with a time limit of two hours per trial to perform a binary classification. We consider (1.0 – AUROC) (area under the ROC curve) as the black-box objective function that needs to be minimized. While the OpenML repositories (Bischl et al. 2017) have many datasets, most of them have a small number of training examples that are insufficient to perform multi-fidelity optimization. We attempt to select the largest possible datasets, ending up with 19 datasets from the OpenML repositories whose data size ranges between 10,885 and 245,057 examples.[4] For a consistent evaluation, we first impute any missing values with the most common value of the corresponding feature and subsequently perform one-hot encoding of the categorical features. We consider a 4-stage pipeline structure which results in a total of 3072 possible pipelines (see Table 1). For each benchmark, all algorithms use the same training and validation sets. In addition, BLDS, Hyperband and DAUB use the identical strategy to increase the subset of training data. With initial experiments using different benchmarks, we set $b = 100$ and $\eta = 2$ for all the algorithms, and $cL/\delta = 1/9600$ for BLDS, where $L = 3072$. Due to the UCB and LCB formulas, note that we do not need to optimize c and δ separately. The exploration parameter for CMAB is set to 0.3.

Results with Algorithm Selection

Figures 3(a)-(f) show the performance of the algorithm selection methods for representative benchmarks. For clarity, we use doubly logarithmic plots. After 10 runs for each algorithm, we compute a median of the objective values and the region within the first and third quartiles. The fixed default parameters from scikit-learn are used for each instantiated pipeline and no HPO is performed. These results clearly demonstrate that BLDS(1) tends to achieve better objective values much more quickly than the other competitors. For example, BLDS(1) outperforms the others for the first 30-500 seconds in many cases including ELECTRICITY and NOMAO.

BLDS(2) tends to catch up with BLDS(1) roughly in 700 seconds, remaining to be slightly better than BLDS(1) for a while (around up to 4500 seconds) until BLDS(1) catches up again. While this indicates a promise of BLDS(2), it does not necessarily outperform BLDS(1) for a short run of the algorithm. We hypothesize that this is due to a much larger number of pipelines needed to be re-trained and evaluated within a discrepancy threshold $disc$. If BLDS(1) finds no pipeline better than the initial one, it examines 26 pipelines within $disc = 1$. BLDS(1) then restarts with a new pipeline

[4]The names and ids of the datasets are ELECTRICITY (151), ADULT (179), MOZILLA4 (1046), JM1 (1053), MAGIC TELESCOPE (1120), CLICK PREDICTION SMALL (1220), BANK MARKETING (1461), NOMAO (1486), SKIN SEGMENTATION (1502), AMAZON EMPLOYEE ACCESS (4135), HIGGS (23512), NUMERAI28.6 (23517), RUN_OR_WALK INFORMATION (40922), APSFAILURE (41138), MINIBOONE (41150), GUILLERMO (41159), RICCARDO (41161), DEFAULT OF CREDIT CARD CLIENTS (42477), and RELEVANT IMAGES DATASET (42680).



(a) MAGICTELESCOPE (b) ELECTRICITY (c) NOMAO

(d) APSFAILURE (e) HIGGS (f) ADULT

Figure 3: Performance of each method for algorithm selection for representative datasets, based on median objective value

randomly initialized, which might sometimes be a good starting point. However, before BLDS(2) restarts search with a new initial pipeline, in the worst case, it needs to examine 272 pipelines within $disc = 2$ which are less similar to the current best one. Therefore, there is a big gap in the local search space between $disc = 1$ and 2. Balancing a better amount of work across different discrepancy values is an important extension to BLDS which remains future work.

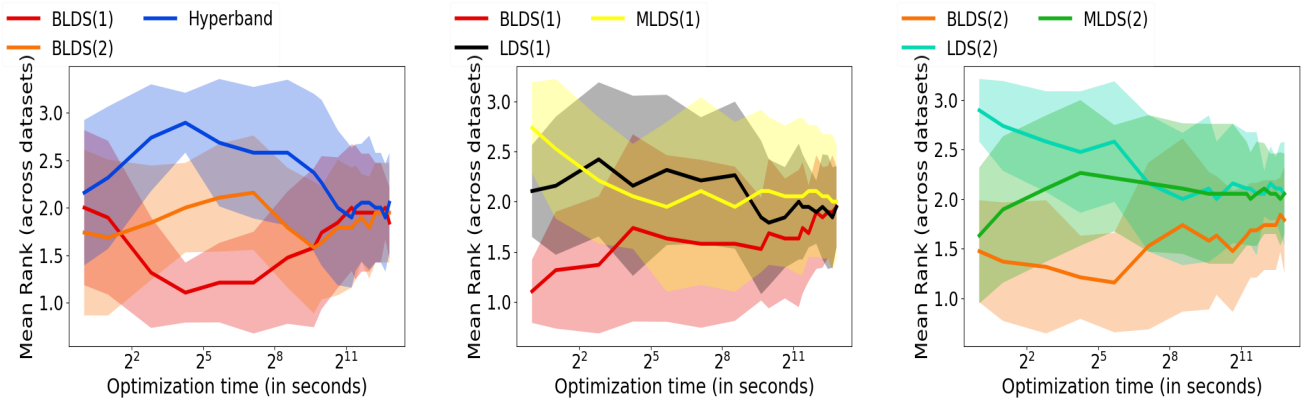
When running for a longer time, the other schemes can catch up with BLDS. At this stage, they can evaluate a sufficient number of pipelines, thus being able to return the objective values competitive to those found by BLDS. For the 9 benchmarks among all, RND evaluates over 1500 pipelines in its 2-hour run, indicating that all algorithms can cover important portions of the 3072 pipelines in such a long runtime.

Figure 4(a) compares the average rank of mean performance of BLDS(1), BLDS(2) and Hyperband, which are the best three algorithms across the 19 benchmarks.[5] An algorithm with a lower rank performs better. Our results clearly show the phenomenon discussed above. Both BLDS(1) and BLDS(2) outperform Hyperband for the first 2000 seconds. Hyperband's performance then becomes similar to that of BLDS. While both BLDS and Hyperband use randomly initialized pipelines, one essential difference is whether to perform local search or not in order to refine the initial pipelines. Our results imply that BLDS' strategy on focusing on the pipelines similar to the current best one is an important factor for finding better pipelines more quickly than Hyperband.

DAUB performs poorly in general. Due to more configurations (i.e., 3072 pipelines) than those in (Sabharwal, Samulowitz, and Tesauro 2016) (only 41 ML classifiers), DAUB suffers from a significant overhead in its bootstrapping step. Even in its pipeline search step, DAUB's linear regression model is not often accurate enough to return an optimized pipeline. DAUB needs to continue search even after a fully-trained pipeline is obtained. DAUB eventually finds an optimized pipeline, but suffers from much slower convergence.

Figures 4(b) and 4(c) compare the average rank of mean performance of BLDS, MLDS and LDS. When the algorithms with the same discrepancy thresholds are compared, our results demonstrate that BLDS outperforms LDS and MLDS, indicating the importance of using our confidence bounds to refine the search. A comparison between LDS and MLDS indicates that introducing the multi-fidelity optimiza-

[5] Since the convergence values are occasionally slightly different, we allow for a tolerance value of 0.001 when calculating each rank.



(a) BLDS versus Hyperband (b) BLDS(1) versus (M)LDS(1) BLDS(2) versus (M)LDS(2)

Figure 4: Comparison against Hyperband, LDS and MLDS, based on average ranking across all datasets

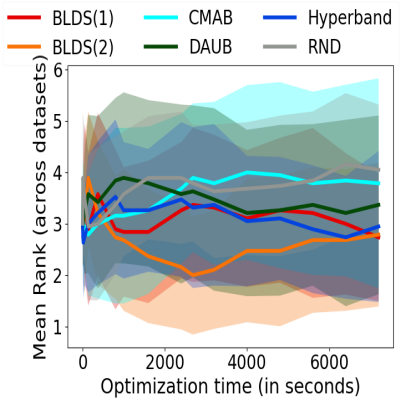


Figure 5: Average ranking across all datasets for each algorithm in a combination with HPO under ADMM

tion tends to yield slightly better results in an early stage until a sufficient number of pipeline evaluations is performed.

Results with Algorithm Selection and HPO

We evaluate the performance of the algorithm selection methods with HPO under the ADMM framework of Liu et al. (2020). There are other approaches that are not based on ADMM, such as Auto-sklearn and TPOT. However, we have decided not to make a direct empirical comparison to these approaches, since our main focus is to develop efficient algorithm selection methods running under ADMM on the basic CASH problem. TPOT addresses the pipeline generation problem that is different from the basic CASH. Auto-sklearn includes meta-learning that remains future work.

Figure 5 shows the average rank of mean performance of each method across 19 benchmarks with 10 runs for each case. As described in (Liu et al. 2020), ADMM starts with 16 iterations with an additive factor 16 until 128 iterations. The figure demonstrates that BLDS performs better than the others. Hyperband catches up with BLDS when more pipelines are evaluated.

ADMM with BLDS(2) converges faster than BLDS(1). We hypothesize that this is because ADMM's time allocation scheme is based not on the actual runtime but *on the number of iterations*. As already discussed, BLDS(2) evaluates a much larger number of pipelines per iteration than that of BLDS(1). This allows ADMM with BLDS(2) to spend much more time in algorithm selection than ADMM with BLDS(1).

Of the 19 benchmarks, BLDS(1) with HPO yields more accurate objective values for 17 benchmarks than BLDS(1) without HPO. BLDS(2) with HPO does so for 15 benchmarks, showing that incorporating HPO generally improves the performance. We conclude that BLDS(2) with HPO is the best choice under the current time allocation scheme.

Conclusion

We introduced BLDS to address the algorithm selection problem. Our results clearly show that BLDS is a well-performing algorithm which tends to converge more quickly than other competing algorithms and that BLDS is a strong candidate to be combined with HPO under ADMM. In future work, we plan to further improve the search performance to be able to deal with large-scale training data as well as more complicated pipeline structures. Possible extensions include a combination with meta-learning, e.g., (Feurer et al. 2015a; Rakotoarison, Schoenauer, and Sebag 2019), and an approach that allows for a more granular control of the local search space. It is also important to have a better theoretical understanding to our MAB strategy by accounting for BLDS' behavior that limits the search space.

Broader Impact

AutoML is an important topic, since it is tedious and time-consuming to manually optimize the pipeline. BLDS can quickly develop more accurate pipelines deployed in real-world applications arising in the society. However, BLDS has not yet addressed the case with the biased dataset. In this case, even if BLDS generates an optimized pipeline, its actual performance might not meet what practitioners expect.

References

Assuncao, F.; Lourenco, N.; Ribeiro, B.; and Machado, P. 2020. Evolution of Scikit-learn Pipelines with Dynamic Structured Grammatical Evolution. In *International Conference on the Application of Evolutionary Computation*, 530–545.

Audibert, J.-Y.; Bubeck, S.; and Munos, R. 2010. Best Arm Identification in Multi-Armed Bandits. *COLT 2010*, 41.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47: 235–256.

Bergstra, J. S.; Bardenet, R.; Bengio, Y.; and Kegl, B. 2011. Algorithms for Hyper-Parameter Optimization. In *NeuIPS*, 2546–2554.

Bischl, B.; Casalicchio, G.; Feurer, M.; Hutter, F.; Lang, M.; Mantovani, R. G.; van Rijn, J. N.; and Vanschoren, J. 2017. OpenML Benchmarking Suites and the OpenML100. https://arxiv.org/abs/1708.03731. Dataset available at https://www.openml.org/.

Chen, B.; Wu, H.; Mo, W.; Chattopadhyay, I.; and Lipson, H. 2018. Autostacker: A Compositional Evolutionary Learning System. In *Genetic and Evolutionary Computation Conference*, 402–409.

de Sa, A.; Freitas, A.; and Pappa, G. 2018. Automated Selection and Configuration of Multi-Label Classification Algorithms with Grammar-Based Genetic Programming. In *International Conference on Parallel Problem Solving from Nature*, 308–320.

de Sa, A.; Pinto, W.; Oliveira, L.; and Pappa, G. 2017. RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines. In *European Conference on Genetic Programming*, 246–261.

Drori, I.; Krishnamurthy, Y.; Rampin, R.; de Paula Lourenco, R.; Ono, J.; Cho, K.; Silva, C.; and Freire, J. 2018. AlphaD3M: Machine Learning Pipeline Synthesis. In *Workshop on AutoML (ICML)*.

Even-Dar, E.; Mannor, S.; and Mansour, Y. 2006. Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems. *Journal of Machine Learning Research*, 7(Jun): 1079–1105.

Falkner, S.; Klein, A.; and Hutter, F. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *ICML*, 1436–1445.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M.; and Hutter, F. 2015a. Efficient and Robust Automated Machine Learning. In *NeurIPS*, 2962–2970.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J. T.; Blum, M.; and Hutter, F. 2015b. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *NeurIPS*, 2962–2970.

Figueroa, R. L.; Zeng-Treitler, Q.; Kandula, S.; and Ngo, L. H. 2012. Predicting Sample Size Required for Classification Performance. *BMC Medical Informatics and Decision Making*, 12(8).

Harvey, W.; and Ginsberg, M. 1995. Limited Discrepancy Search. In *IJCAI*, 607–613.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *International Conference on Learning and Intelligent Optimization*, 507–523.

Jamieson, K. G.; and Talwalkar, A. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS*, 240–248.

Karnin, Z.; Koren, T.; and Somekh, O. 2013. Almost Optimal Exploration in Multi-Armed Bandits. In *ICML*, 1238–1246.

Katz, M.; Ram, P.; Sohrabi, S.; and Udrea, O. 2020. Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning. In *ICAPS*, 403–411.

Klein, A.; Falkner, S.; Bartels, S.; Hennig, P.; and Hutter, F. 2017. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In *AISTATS*, 528–536.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer.

Kohavi, R. 1995. A study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, 1137–1145.

Korf, R. E. 1996. Improved Limited Discrepancy Search. In *AAAI*, 286–291.

Koshute, P.; Zook, J.; and McCulloh, I. 2021. Recommending Training Set Sizes for Classification. https://arxiv.org/abs/2102.09382.

Kotthoff, L.; Thornton, C.; Hoos, H.; Hutter, F.; and Leyton-Brown, K. 2017. Auto-WEKA 2.0: Automatic Model Selection and Hyperparameter Optimization in WEKA. *Journal of Machine Learning Research*, 18(1): 826–830.

Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; and Talwalkar, A. 2018. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18: 1–52.

Liu, S.; Ram, P.; Vijaykeerthy, D.; Bouneffouf, D.; Bramble, G.; Samulowitz, H.; Wang, D.; Conn, A.; and Gray, A. 2020. An ADMM Based Framework for AutoML Pipeline Configuration. In *AAAI*, 4892–4899. Their supplementary material available at https://arxiv.org/pdf/1905.00424.pdf.

Lu, Z.; Chen, L.; Chiang, C.-K.; and Sha, F. 2019. Hyperparameter Tuning under a Budget Constraint. In *IJCAI*, 5744–5750.

Marinescu, R.; Kishimoto, A.; Ram, P.; Rawat, A.; Wistuba, M.; Palmes, P.; and Botea, A. 2021. Searching for Machine Learning Pipelines Using a Context-Free Grammar. In *AAAI*, 8902–8911.

Mohr, F.; and van Rijn, J. N. 2021. Towards Model Selection using Learning Curve Cross-Validation. In *8th ICML Workshop on Automated Machine Learning*.

Mohr, F.; and Wever, M. 2021. Replacing the Ex-Def Baseline in AutoML by Naive AutoML. In *8th ICML Workshop on Automated Machine Learning*.

Mohr, F.; Wever, M.; and Hullermeier, E. 2018. ML-Plan: Automated Machine Learning via Hierarchical Planning. *Machine Learning*, 107(1): 1495–1515.

Olson, R.; Bartley, N.; Urbanowicz, R.; and Moore, J. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Genetic and Evolutionary Computation Conference*, 485–492.

Pedregosa, F.; Varoquaux, G.; and Gramfort, A. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(1): 2825–2830.

Rakotoarison, H.; Schoenauer, M.; and Sebag, M. 2019. Automated Machine Learning with Monte-Carlo Tree Search. In *IJCAI*, 3296–3303.

Sabharwal, A.; Samulowitz, H.; and Tesauro, G. 2016. Selecting Near-Optimal Learners via Incremental Data Allocation. In *AAAI*, 2007–2015.

Shahriari, B.; Swersky, K.; Wang, Z.; Adams, R. P.; and Freitas, N. D. 2016. Taking the Human out of the Loop: A Review of Bayesian Optimization. In *Proceedings of the IEEE*, 148–175.

Sun, X.; Lin, J.; and Bischl, B. 2020. ReinBo: Machine Learning Pipeline Search and Configuration with Bayesian Optimization Embedded Reinforcement Learning. In *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019 (Part I)*, volume 1167 of *Communications in Computer and Information Science*, 68–84. Springer.