Journal of Machine Learning Research Volume (Year) Pages Submitted 4/19; Published Date Published

Survey on Automated Machine Learning

Marc-André Zöller M.ZOELLER@USU.DE
USU Software AG
Rüppurrer Str. 1, Karlsruhe, Germany

Marco F. Huber MARCO.HUBER@IEEE.ORG
Institute of Industrial Manufacturing and Management IFF, University of Stuttgart
Center for Cyber Cognitive Intelligence CCI, Fraunhofer IPA
Nobelstr. 12, Stuttgart, Germany

Editor:

Abstract

Machine learning has become a vital part in many aspects of our daily life. However, building well performing machine learning applications requires highly specialized data scientists and domain experts. Automated machine learning (AutoML) aims to reduce the demand for data scientists by enabling domain experts to automatically build machine learning applications without extensive knowledge of statistics and machine learning. In this survey, we summarize the recent developments in academy and industry regarding AutoML. First, we introduce a holistic problem formulation. Next, approaches for solving various subproblems of AutoML are presented. Finally, we provide an extensive empirical evaluation of the presented approaches on synthetic and real data.

Keywords: automated machine learning, AutoML, machine learning, hyperparameter optimization, survey

1. Introduction

In recent years machine learning (ML) is becoming ever more important: automatic speech recognition, self-driving cars or predictive maintenance in industry 4.0 are build upon ML. ML is nowadays able to beat human beings in tasks often described as too complex for computers, e.g., AlphaGO (Silver et al., 2017) was able to beat the human champion in GO. All these examples are powered by extremely specialized and complex ML pipelines.

In order to build such an ML pipeline, a highly trained team of human experts is necessary: data scientists have profound knowledge of ML algorithms and statistics; domain experts often have a longstanding experience within a specific domain. Together, those human experts can build a sensible ML pipeline containing specialized data preprocessing, domain-driven meaningful feature engineering and fine-tuned models leading to astonishing predictive power. Usually, this process is a very complex task, performed in an iterative manner with trial and error. As a consequence, building good ML pipelines is a long and expensive endeavor.

According to the *no free lunch* theorem (Wolpert and Macready, 1997) it is impossible that a single optimization algorithm is universally superior to any other optimization algorithm. This implies that no universally superior ML pipeline for all ML tasks exists. Consequently, it is not possible to transfer a well performing ML pipeline to a new domain

and still yield outstanding results. Instead, a new ML pipeline has to be constructed for each new data set to obtain optimal results. However, manually building a specialized ML pipeline for each and every problem is very time consuming and therefore expensive. As a consequence, practitioners often use a suboptimal default ML pipeline.

AutoML aims to improve the current way of building ML applications by automation. ML experts can profit from AutoML by automating tedious tasks like hyperparameter optimization (HPO) leading to a higher efficiency. Domain experts can be enabled to build ML pipelines on their own without having to rely on a data scientist.

It is important to note that AutoML is no new trend. Starting from the 1990s commercial solutions offered automatic HPO for selected classification algorithms via grid search (Dinsmore, 2016). In 2004, the first efficient strategies for HPO have been proposed. For limited settings, e.g., tuning C and γ of a support-vector machine (SVM) (Chen et al., 2004), it was proven that guided search strategies yield better results than grid search in less time. Also in 2004, the first approaches for automatic feature selection have been published (Samanta, 2004). *Full model selection* (Escalante et al., 2009) was the first attempt to automatically build a complete ML pipeline by simultaneously selecting a preprocessing, feature selection and classification algorithm while tuning the hyperparameters of each method. Testing this approach on various data sets, the potential of this domain-agnostic method was proven (Guyon et al., 2008). Starting from 2011, many different methods of applying Bayesian optimization for hyperparameter tuning (Bergstra et al., 2011; Snoek et al., 2012) and model selection (Thornton et al., 2013) have been proposed. In 2015, the first method for automatic feature engineering without domain knowledge was proposed (Kanter and Veeramachaneni, 2015). Building arbitrary shaped pipelines is possible since 2016 (Olson and Moore, 2016). In 2017 and 2018 the topic AutoML received a lot of attention in media (Google, 2019) with the release of commercial AutoML solutions from various global players (Golovin et al., 2017; Clouder, 2018; Baidu, 2018). Simultaneously, research in the area of AutoML gained significant traction leading to many performance improvements. Recent methods are able to reduce the runtime of AutoML procedures from several hours to mere minutes (Hutter et al., 2018).

This paper reviews and summarizes research concerning the automation of any aspect of an ML pipeline: determining the pipeline shape, selecting an ML algorithm for each stage in a pipeline and tuning each algorithm. Furthermore, the most popular open-source AutoML frameworks are analyzed. This survey focuses on classical machine learning and does **not** consider neural networks while still many of the ideas can be transferred to them. A comprehensive overview of neural networks, especially network architecture search, is provided by Baker et al. (2016) and Zoph and Le (2017).

The contributions of this paper are as following:

- We introduce a mathematical formulation covering the complete procedure of automatic ML pipeline creation.
- To the best of our knowledge, this paper is the first survey covering AutoML techniques for each step of building an ML pipeline.
- An extensive empirical evaluation of all presented algorithms is performed on synthetic and real data. These experiments allow a fair comparison of state-of-the-art approaches under identical conditions.

In doing so, readers will get a comprehensive overview of state-of-the-art AutoML algorithms. All important stages of building an ML pipeline automatically are introduced and existing approaches are evaluated. This allows revealing the limitations of current approaches and rising open research questions.

In Section 2 a mathematical sound formulation of the automatic construction of ML pipelines is given. Section 3 presents different strategies for determining a pipeline structure. Various approaches for ML model selection and HPO are theoretically explained in Section 4. Next, methods for automatic data cleaning (Section 5) and feature engineering (Section 6) are introduced. Measures for improving the performance of the generated pipelines as well as decreasing the optimization runtime are explained in Section 7. Section 8 provides a short introduction to existing AutoML implementations and frameworks. These frameworks are evaluated on various data sets in Section 9. Finally, opportunities for further research are presented in Section 10 followed by a short conclusion in Section 11.

2. Problem Formulation

An ML pipeline is a sequential combination of various algorithms that transforms a feature vector $\vec{x} \in \mathbb{X}^d$ into a target value $y \in \mathbb{Y}$, e.g., a class label for a classification problem. Without loose of generality these algorithms can be grouped in three sets: *data cleaning*, *feature engineering*, and *modeling*. For each set, a fixed number of algorithms $A_{\text{cleaning}} = \{A^{(1)}_{\text{cleaning}}, \ldots, A^{(R)}_{\text{cleaning}}\}$ is available, with A_{feature} and A_{model} defined accordingly. For simplicity the subscripts are ignored where possible. Let the set of all algorithms $\mathcal{A}$ be defined as $\mathcal{A} = \{A_{\text{cleaning}} \cup A_{\text{feature}} \cup A_{\text{model}}\}$. Furthermore, each algorithm $A^{(i)}$ is configured by a vector of hyperparameters $\vec{\lambda}^{(i)}$ from the domain $\Lambda^{(i)}$.

Definition 1 (Machine Learning Pipeline) *Let a pipeline structure $g \in G$ be an arbitrary directed acyclic graph (DAG). Each node represents a data preprocessing, feature engineering, or modeling algorithm. The edges represent the flow of an input data set through the different algorithms. An ML pipeline is a triple $(g, \vec{A}, \vec{\lambda})$ with $\vec{A} \in \mathcal{A}^+$ a vector consisting of the selected algorithm for each node and $\vec{\lambda}$ a vector of the hyperparameters for each selected algorithm. The pipeline is denoted as $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$.*

The triple $(g, \vec{A}, \vec{\lambda})$ is called a *configuration*. Given a pipeline $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$ it is important to assess its performance for a given data set in order to build pipelines with a low generalization error.

Definition 2 (Pipeline Performance) *Let a pipeline $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$ be given. For $i = 1, \ldots, n$, let $\vec{x}_i \in \mathbb{X}^d$ denote a feature vector and $y_i \in \mathbb{Y}$ the corresponding target value. Given a data set $D = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$ and a loss metric $\mathcal{L}(\cdot, \cdot)$, the performance π of $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$ is calculated as*

$$\pi = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y_i),$$

with $\hat{y}_i = \mathcal{P}_{g,\vec{A},\vec{\lambda}}(\vec{x}_i)$ being the predicted output of $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$ given the sample $\vec{x}_i$.

This definition is limited to supervised learning but can be easily extended for unsupervised and reinforcement learning by choosing appropriate loss functions.

A loss function $\mathcal{L} : \mathbb{Y} \times \mathbb{Y} \to \mathbb{R}$ is a numeric function assessing the performance of a given model. In general, the loss function depends on the underlying problem: a common loss function for regression is the mean squared error (Wang and Bovik, 2009) and hinge loss is often used for classification (Rosasco et al., 2004).

Let an *ML task* be defined by a given data set, loss function and a specific ML problem type, e.g., classification or regression. The problem of generating an ML pipeline for a given ML task can be split into three tasks: at first, the structure of the pipeline has to be determined, for example selecting how many data preprocessing and feature engineering steps are necessary, how the data flows through the pipeline and how many models have to be trained. Next, for each of these steps a specific algorithm has to be selected, e.g., a classification model can be trained using a SVM or a decision tree. Finally, for each selected algorithm its corresponding hyperparameters have to be selected. The goal of each step is to obtain a pipeline with a minimal loss on a validation set. However, all three steps have to be completed to actually evaluate the pipeline performance.

Definition 3 (Pipeline Creation Problem) *Let a set of algorithms $\mathcal{A}$ with an according domain of hyperparameters $\Lambda^{(\cdot)}$ be given. Furthermore, let a training data set $D_{train} = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$ be split into K cross-validation folds $\{D_{valid}^{(1)}, \ldots, D_{valid}^{(K)}\}$ and $\{D_{train}^{(1)}, \ldots, D_{train}^{(K)}\}$ such that $D_{train}^{(i)} = D_{train} \setminus D_{valid}^{(i)}$. Then, the pipeline creation problem consists of finding a pipeline structure together with a joint algorithm and hyperparameter selection that minimizes the loss*

$$g^\star, \vec{A}^\star, \vec{\lambda}^\star \in \underset{g \in G, \vec{A} \in \mathcal{A}^+, \vec{\lambda} \in \Lambda}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}\left(\mathcal{P}_{g,\vec{A},\vec{\lambda}}\left(D_{train}^{(i)}\right),\ D_{valid}^{(i)}\right) \tag{1}$$

with $\mathcal{L}\left(\mathcal{P}_{g,\vec{A},\vec{\lambda}}\left(D_{train}\right),\ D_{valid}\right)$ denoting the performance of the pipeline $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$ created from the training set D_{train} being evaluated on the validation set D_{valid}.

Even though Equation (1) formulates the pipeline creation problem as a minimization, loss functions like accuracy, F1 or R2 score can still be modeled by negating the calculated loss. Using Equation (1), the pipeline creation problem is formulated as a black box optimization problem.

Finding the global optimum in equations like (1) has been the subject of decades of study (Snyman, 2005). Many different algorithms have been proposed to efficiently solve specific problem instances, for example convex optimization for convex objective functions f with convex domains. To use these methods the features and shape of the underlying objective function—in this case the loss $\mathcal{L}$—have to be known to select applicable solvers. In general, it is not possible to predict any properties of the loss function or even formulate it as closed-form expression, as it depends on the training data. Consequently, efficient solvers, like convex or gradient-based optimization, cannot be used for Equation (1) (Luo, 2016).

Human ML experts usually solve the pipeline creation problem in an iterative manner: At first a simple pipeline structure with standard algorithms and default hyperparameters is selected. Next, the pipeline structure is adapted, potentially new algorithms are selected and hyperparameters are refined. This procedure is repeated until the overall performance

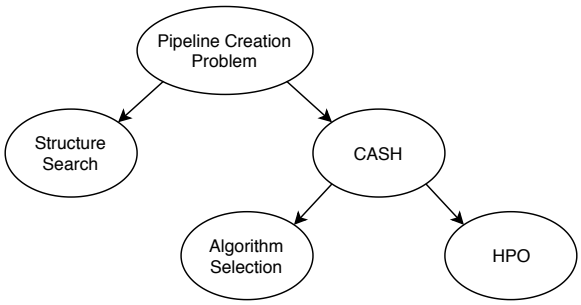


Figure 1: Subproblems of the pipeline creation problem.

is sufficient. In contrast, most current state-of-the-art algorithms solve the pipeline creation problem in two distinct stages. At first, the pipeline shape is determined, this stage is introduced in detail in Section 3. Next, the actual algorithms $\vec{A}$ and their according hyperparameters $\vec{\lambda}$ are selected. This stage is introduced in Section 4. Figure 1 shows a schematic representation of the different optimization problems for the automatic creation of ML pipelines. Solutions for each subproblem are presented in the following sections.

3. Pipeline Structure Creation

The first task for building an ML pipeline is creating the pipeline structure. This topic has received a lot of attention in the context of designing neural networks referred to as *architecture search*, e.g., (Zoph and Le, 2017; Liu et al., 2017). Regarding classical machine learning, this topic was tackled for various specific problems like natural language processing, e.g., (Agerri et al., 2014). Surprisingly, basically no publications exist treating general pipeline construction. Yet, common best practices suggest a basic ML pipeline layout as displayed in Figure 2 (Kégl, 2017; Ayria, 2018; Zhou, 2018). At first, the input data is cleaned in multiple distinct steps, like imputation of missing data and one-hot encoding of categorical input. Next, relevant features are selected and new features created in a feature engineering stage. This stage highly depends on the underlying domain. Finally, a single model is trained on the previously selected features.

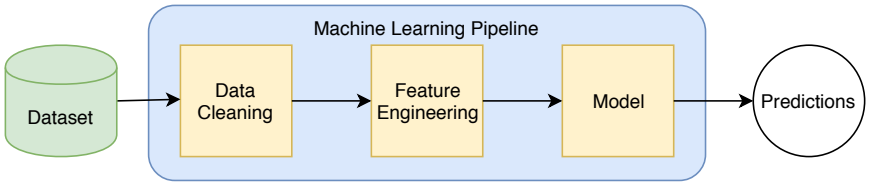


Figure 2: Prototypical ML pipeline. First, the input data is cleaned; next, features are extracted. Finally, the transformed input is passed through an ML model to create predictions.

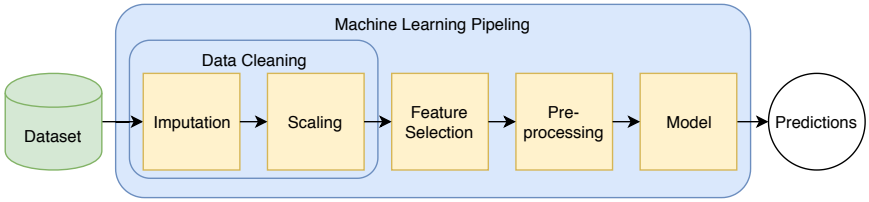


Figure 3: Fixed ML pipeline used by most AutoML frameworks. Minor differences exist regarding the implemented data cleaning steps.

3.1 Fixed Shape

Many AutoML frameworks do not solve the structure selection because they are preset to the fixed pipeline shape displayed in Figure 3, e.g., (Komer et al., 2014; Feurer et al., 2015a; Swearingen et al., 2017; Parry, 2019; McGushion, 2019). Resembling the best practice pipeline closely, the pipeline is a linear sequence of multiple data cleaning steps, a feature selection step, one variable preprocessing step and exactly one modeling step. The preprocessing step chooses at most one algorithm from a set of well known algorithms, e.g., various matrix decomposition algorithms. Regarding data cleaning, the pipeline shape differs. Yet, often the two steps imputation and scaling are implemented.

By using a pipeline with a fixed shape, the complexity of determining a graph structure g is completely eliminated and the pipeline creation problem is reduced to selecting a preprocessing and modeling algorithm. Even though this approach greatly reduces the complexity of the pipeline creation problem, it leads to inferior pipeline performances for complex data sets. Yet, for many problems with high quality training data a simple pipeline structure may still be sufficient.

3.2 Variable Shape

Data science experts usually build highly specialized pipelines for a given ML task to obtain the best results. Fixed shaped ML pipelines lack this flexibility to adapt to a specific task. Several approaches for automatically building flexible pipelines exist that are all based on the same principal ideas: a pipeline consists of a set of ML primitives—namely the basic algorithms $\mathcal{A}$—, an operator to clone a data set and an operator to combine multiple data sets—referred to as *data set duplicator* and *feature union*. The data set duplicator is used to create parallel paths in the pipeline; parallel paths can be joined via a feature union. A pipeline using all these operators is displayed in Figure 4. In the following multiple approaches to combine these primitives to a complete pipeline are presented.

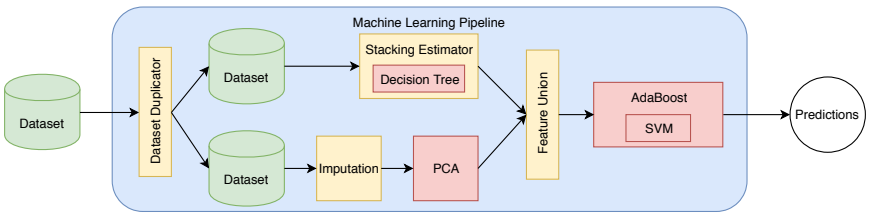


Figure 4: Specialized ML pipeline for a specific ML task.

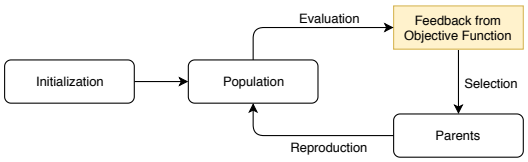


Figure 5: Schematic procedure of genetic programming. An initial population is randomly created. The complete population is evaluated against an objective function and the best performing individuals are selected. These parents are used to generate a new population.

3.2.1 Genetic Programming

The first method to build flexible ML pipelines automatically was introduced by Olson and Moore (2016); Olson et al. (2016a) and is based on genetic programming (Banzhaf et al., 1997). Genetic programming is an iterative, domain-agnostic optimization method derived from biological evolution. A set of possible solutions for a problem—called a *population*—is evaluated using a fitness function. Based on each individual's performance, a certain proportion of their *genetic material* is transferred to the next generation. Bad performing individuals become extinct while well performing individuals produce offsprings. A descendant is either created via crossover of two parents or by mutating a single parent. This procedure is repeated for a fixed number of generations with a fixed number of individuals per generation; the first generation is usually randomly generated. In the end, the individual with the best performance is used as the solution. This procedure is visualized in Figure 5.

Genetic programming has been used for automatic program code generation for a long time (Poli et al., 2008). Yet, the application to pipeline structure selection is quite recent. In the beginning, random pipeline structures are created and evaluated on a training data set. The best performing pipelines are selected for creating the next generation. Two individuals are combined by selecting sub-graphs of the pipelines and combining these sub-graphs to a new graph. Mutation can be implemented by adding a random node to the graph, removing a randomly selected node or replacing a randomly selected node with a new one. This way arbitrary shaped pipelines can be generated. However, genetic programming is not very efficient concerning the number of objective function evaluations (Cooper and Hinde, 2003). As each objective function evaluation is relatively expensive, this approach requires a quite long optimization period.

3.2.2 Hierarchical Planning

A more directed approach is based on hierarchical task networks (HTNs) (Ghallab et al., 2004). HTNs are a method from automated planning that recursively partition a complex problem into easier subproblems. These subproblems are again decomposed until only atomic terminal operations are left. This procedure can be visualized as a graph structure. Each node represents a (potentially incomplete) pipeline; each edge the decomposition of a complex step into sub-steps. When all complex problems are replaced by ML primitives, an ML pipeline is created. Using this abstraction, the problem of finding an ML pipeline structure is reduced to finding the best leaf node in the graph (Mohr et al., 2018).

3.2.3 Self-Play

Self-play (Lake et al., 2017) is a reinforcement learning strategy that has received a lot of attention lately due to the recent successes of AlphaZero (Silver et al., 2017). Instead of learning from a fixed data set, the algorithm creates new training examples by playing against itself. Pipeline structure search can also be considered as a game (Drori et al., 2018): an ML pipeline and the training data set represent the current board state s; at each step the player can choose between the three actions a adding, removing or replacing a single element in the pipeline; the loss of the pipeline is used as a score $\nu(s)$. In an iterative procedure, a neural network is used to evaluate a pipeline s_i by predicting its score $\nu(s_i)$ and probabilities which action to chose in this state $P(s_i, a)$. Without training, these predictions are basically random. Therefore, the predictions are passed to a Monte Carlo tree search (Browne et al., 2012). In this tree, each node represents a pipeline structure and each edge a possible action. Based on the state and action probabilities, a node is selected. If the node was not visited before, the according pipeline is evaluated on the data set and the performance is used to update the neural network. The next state s_{i+1} is selected as the node with the lowest predicted loss. These three steps are repeated until the training budget is exhausted. A common drawback of self-play approaches is the low convergence speed (Brandt et al., 2010) making this approach rather unsuited for AutoML.

4. Algorithm Selection and Hyperparameter Optimization

Let a shape g, a loss function $\mathcal{L}$, a training set D_{train} and a validation set D_{valid} be given. For each node in g an algorithm has to be selected and configured via hyperparameters. This section introduces various methods for algorithm selection and configuration.

A notion first introduced by Thornton et al. (2013) and since then adopted by many others is the combined algorithm selection and hyperparameter optimization (CASH) problem. Instead of selecting an algorithm first and optimizing its hyperparameters later, both steps are executed simultaneously. This problem is formulated as a black box optimization problem leading to a minimization problem quite similar to the pipeline creation problem in Equation (1)

$$A^\star, \vec{\lambda}^\star \in \underset{A^{(j)} \in \mathcal{A}, \vec{\lambda} \in \Lambda^{(j)}}{\arg\min} \mathcal{L}\left(A^{(j)}_{\vec{\lambda}}\left(D_{\text{train}}\right), D_{\text{valid}}\right). \tag{2}$$

with $\mathcal{L}\left(A^{(j)}_{\vec{\lambda}}\left(D_{\text{train}}\right), D_{\text{valid}}\right)$ being the performance of $A^{(j)}$ configured by $\vec{\lambda}$ for a given D_{train} and D_{valid}.

It is important to note that Equation (2) is not easily solvable as the search space is quite large and complex. For example, consider selecting a classification algorithm: scikit-learn (Pedregosa et al., 2011) currently implements 30 different classification algorithms with a total number of over 250 hyperparameters. Some of the hyperparameters are categorical, like the splitting criterion used for decision trees, others are integer-valued, like the number of neighbors in k-nearest neighbors, and some are real-valued, like the slack variable for SVMs. Furthermore, like the pipeline creation problem, the loss function is usually non-smooth and derivative-free.

An exhaustive search of this space is not feasible. Assuming each continuous parameter is discretized to at most 5 values and assuming that each integer-valued and categorical

parameter also comprises 5 values, the transformed search space contains more than 5^{250} different realizations. For any realistic D_{train}, the time to train 5^{250} classifiers is infeasible. However, the search space can be drastically reduced by modeling correlations between different hyperparameters.

Let the choice which algorithm to use be treated as an additional categorical meta-hyperparameter λ_r. Then the complete hyperparameter space for n algorithms can be defined as

$$\Lambda = \Lambda^{(1)} \times \dots \Lambda^{(n)} \times \lambda_r$$

referred to as the *configuration space*.

The hyperparameter λ_r has the strongest correlation with all other hyperparameters. If for example the ith algorithm is selected only $\Lambda^{(i)}$ is relevant as all other hyperparameters do not influence the result. Therefore, $\Lambda^{(i)}$ depends on $\lambda_r = i$. Following Hutter et al. (2009); Thornton et al. (2013); Swearingen et al. (2017) the hyperparameters $\lambda \in \Lambda^{(i)}$ can be aggregated in two groups. Mandatory hyperparameters always have to be present, for example the slack variable of a SVM. Conditional hyperparameters depend on the selected value of another hyperparameter, e.g., a SVM only requires a degree for polynomial kernels. A hyperparameter λ_i is conditional on another hyperparameter λ_j, if and only if λ_i is relevant when λ_j takes values from a specific set $V_i(j) \subset \Lambda_j$.

Using this notation, the configuration space can be interpreted as a tree as visualized in Figure 6. λ_r represents the root node with a child node for each algorithm. Each algorithm has the according mandatory hyperparameters as child nodes, all conditional hyperparameters are children of one mandatory hyperparameter. This tree structure can be used to significantly reduce the search space, resulting in the final minimization problem

$$\vec{\lambda}^{\star} \in \arg\min_{\vec{\lambda} \in \Lambda} \mathcal{L}\left(A_{\vec{\lambda}}\left(D_{\text{train}}\right), D_{\text{valid}}\right). \tag{3}$$

For simplicity, let $f(\lambda) = \mathcal{L}\left(A_{\vec{\lambda}}\left(D_{\text{train}}\right), D_{\text{valid}}\right)$ be denoted as the *objective function*. The rest of this section introduces different optimization strategies to solve Equation (3).

4.1 Grid Search

The first approach proposed to systematically explore the configuration space was grid search. As the name implies, grid search creates a grid of configurations and evaluates all of them. Therefore, each continuous hyperparameter is discretized into k (logarithmic) equidistant values $\lambda_{1:k}^{(i)}$; for categorical hyperparameters each value is used (Hsu et al., 2003). By using the Cartesian product of the discretized hyperparameters, a finite search space

$$\Lambda_{GS} = \lambda_{1:k}^{(1)} \times \lambda_{1:k}^{(2)} \times \dots \times \lambda_{1:k}^{(n)}$$

is created. Figure 7 depicts via grid search selected configurations for two hyperparameters.

Even though grid search is easily implemented and parallelized (Bergstra and Bengio, 2012), it has some major drawbacks. This basic algorithm does not scale well for large configuration spaces, as the number of function evaluations grows exponentially with the number of hyperparameters (LaValle et al., 2004). This phenomenon is known as *curse of dimensionality* (Friedman, 1997). Furthermore, the hierarchical hyperparameter structure is not considered, leading to redundant configurations.

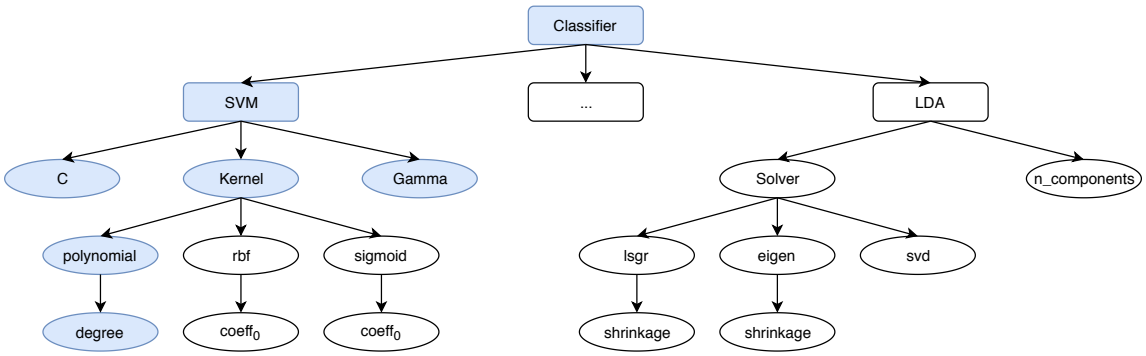


Figure 6: Incomplete representation of the structured configuration space for selecting and tuning a classification algorithm. Rectangle nodes represent the selection of an algorithm. Ellipse nodes represent tunable hyperparameters. Highlighted in blue is an active set of configurations to select and configure a SVM with a polynomial kernel.

In the classical version, grid search does not exploit knowledge of well performing regions. This drawback is partially eliminated by *contradicting* grid search (Hsu et al., 2003; Hesterman et al., 2010). At first, a coarse grid is fitted, next a finer grid is created centered around the best performing configuration. This iterative procedure is repeated k times converging to a local minimum.

4.2 Random Search

Another widely-known approach is random search (Anderson, 1953). A candidate configuration is generated by randomly choosing a value for each hyperparameter independently of all others. Conditional hyperparameters can be implicitly handled by traversing the hierarchical dependency graph. This procedure is repeated k times. Figure 7 depicts configurations selected by random search for two hyperparameters.

Random search is straightforward to implement and parallelize and well suited for gradient-free functions with many local minima (Solis and Wets, 1981). With $k \to \infty$, random search is able to find a solution arbitrary close to the global optimum. Even though, the convergence speed is faster than grid search (Bergstra and Bengio, 2012), still many function evaluations are necessary as no knowledge of well performing regions is exploited. As function evaluations, i.e., training a model, are very expensive, random search requires a long optimization period.

4.3 Sequential Model-Based Optimization

The CASH problem can be treated as a regression problem: the loss function can be approximated using standard regression methods based on the tested hyperparameter configurations. This concept is captured by sequential model-based optimization (SMBO) (Bergstra et al., 2011; Hutter et al., 2011; Bergstra et al., 2013) displayed in Algorithm 1. The loss function $f(\lambda)$ is complemented by a probabilistic regression model M that acts as a sur-

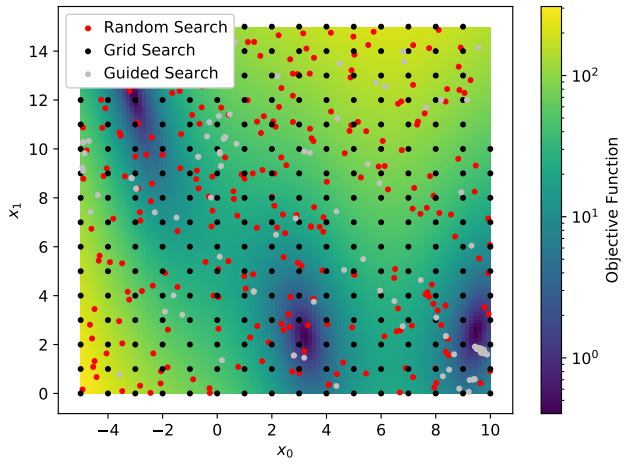


Figure 7: Tested configurations of different search strategies for minimizing the Branin function (see Section 9.2). Grid search tests configurations uniformly distributed over the two dimensional search space. Random search selects configurations at random. Guided search strategies exploit knowledge of well performing regions testing more configurations close to local minima.

rogate for f. The surrogate model M allows predicting the performance of an arbitrary configuration λ without evaluating the demanding objective function f. M is built using all so-far observed performances $D_{1:n} = \{(\lambda_1, f(\lambda_1)), \ldots, (\lambda_n, f(\lambda_n))\}$ (line 3) and is used to sequentially create new configurations (line 4). These new configurations are obtained using a cheap *acquisition function*. Each suggested configuration is evaluated on the objective function f (line 5) and the result added to $D_{1:n}$ (line 6). These steps are repeated until a fixed budget T—usually either a fixed number of iterations or a time limit—is exhausted. The initialization (line 1) is often implemented by selecting a small number of random configurations that are evaluated. Figure 8 shows the complete SMBO procedure.

Algorithm 1 Sequential Model-Based Optimization

Input: f, Λ, T

1: $D \leftarrow$ InitSamples(f, Λ)
2: **for** $i \leftarrow 1$ to T **do**
3: $\quad M \leftarrow$ FitModel(D)
4: $\quad \lambda_i \leftarrow$ SelectConfiguration(M, Λ)
5: $\quad y_i \leftarrow f(\lambda_i)$ ▷ *expensive step*
6: $\quad D \leftarrow D \cup (\lambda_i, y_i)$
7: **end for**

Even though fitting a model and selecting a configuration introduces an computational overhead, the probability of testing bad performing configurations can be significantly lowered. As the actual function evaluation is usually way more expensive then these additional steps, better performing configurations can be found in a shorter time span in comparison to random or grid search.

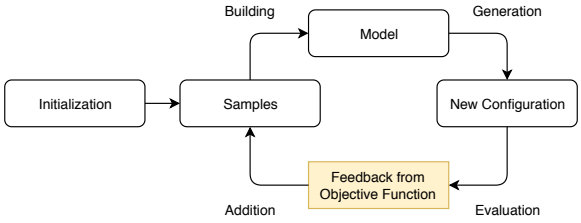


Figure 8: Schematic procedure of SMBO. During the initialization a set of configuration and score tuples is created. These samples are used to create a regression model of the objective function. Next, a new configuration is selected and evaluated by the objective function. Finally, the new tuple is added to the set of samples.

To actually implement the surrogate model fitting and configuration selection (line 3 and 4), Bayesian optimization is used. Bayesian optimization (Brochu et al., 2010) is an iterative optimization framework being well suited for expensive objective functions. Based on previous observations $D_{1:n}$, a probabilistic model of the objective function f is obtained using Bayes' theorem

$$P\left(f \mid D_{1:n}\right) \propto P\left(D_{1:n} \mid f\right) P(f). \tag{4}$$

In the following, the index of $D_{1:n}$ is ignored for simplicity. Bayesian optimization is very efficient concerning the number of objective function evaluations (Brochu et al., 2010) as an acquisition function is used to determine the next configuration $\lambda_{n+1} \in \Lambda$ to evaluate. The acquisition function automatically handles the trade-off between exploration and exploitation: new regions with a high uncertainty are explored, preventing the optimization being stuck in a local minimum; well performing regions with a low uncertainty are exploited converging to a local minimum (Brochu et al., 2010). Many different acquisition functions exist but *expected improvement* (Mockus et al., 1978) is one of the most common (Brochu et al., 2010). The expected improvement is defined as

$$\begin{aligned} EI(\lambda) &= \mathbb{E}\left[\max\left(0, f' - f(\lambda)\right) \mid D\right] \\ &= \int_{-\infty}^{f'} \left(f' - f\right) \cdot p\left(f \mid D\right) \mathrm{d}f \end{aligned} \tag{5}$$

with $f' = \arg\min_{\lambda_i \in D} f(\lambda_i)$ the best observed value so far. To actually propose a new configuration (line 4 in Algorithm 1), the point with the highest expected improvement is selected by

$$\lambda_{n+1} = \arg\max_{\lambda \in \Lambda} \; EI(\lambda). \tag{6}$$

Many different methods exist to solve Equation (6). Their introduction is postponed to Section 8.

The surrogate model M (line 3 in Algorithm 1) corresponds to the posterior $P\left(f \mid D\right)$ in Bayesian optimization. As previously mentioned the characteristics and shape of the loss function are in general unknown. Therefore, the posterior has to be a non-parametric model. The rest of this section introduces non-parametric models used by various AutoML implementations.

4.3.1 Gaussian Process

The traditional surrogate model for Bayesian optimization are Gaussian processes (Rasmussen and Williams, 2006). The key idea is that any objective function f can be modeled using an infinite dimensional Gaussian distribution. Therefore, a Gaussian process is defined by a mean function $m(\lambda)$ and a covariance function $k(\lambda, \lambda')$

$$\begin{aligned} m(\lambda) &= \mathbb{E}\left[f(\lambda)\right] \\ k(\lambda, \lambda') &= \mathbb{E}\left[\left(f(\lambda) - m(\lambda)\right) \cdot \left(f(\lambda') - m(\lambda')\right)\right]. \end{aligned} \tag{7}$$

Even though a Gaussian process is a non-parametric model, it still requires hyperparameters itself, namely the selection of m and k. A common approach is selecting $m(\lambda) = \vec{0}$ for all $\lambda \in \Lambda$. The choice of the covariance function depends on the underlying data and influences the overall performance significantly (Rasmussen and Williams, 2006).

To actually predict the performance of a new configuration λ, the covariance matrix of all previously seen observations has to be calculated as

$$K = \begin{bmatrix} k(\lambda_1, \lambda_1) & k(\lambda_1, \lambda_2) & \dots & k(\lambda_1, \lambda_n) \\ k(\lambda_2, \lambda_1) & k(\lambda_2, \lambda_2) & \dots & k(\lambda_2, \lambda_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(\lambda_n, \lambda_1) & k(\lambda_n, \lambda_2) & \dots & k(\lambda_n, \lambda_n) \end{bmatrix}.$$

Given a new configuration λ_{n+1}, a Gaussian process predicts the model performance by

$$\begin{bmatrix} \vec{f}(\lambda) \\ f(\lambda_{n+1}) \end{bmatrix} \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} K & K_{n+1}^T \\ K_{n+1} & k(\lambda_{n+1}, \lambda_{n+1}) \end{bmatrix}\right)$$

with $K_{n+1} = \begin{bmatrix} k(\lambda_1, \lambda_{n+1}) & k(\lambda_2, \lambda_{n+1}) & \dots & k(\lambda_n, \lambda_{n+1}) \end{bmatrix}$ and $\mathcal{N}$ representing a normal distribution. The actual model performance is predicted by

$$f(\lambda_{n+1}) \mid \vec{f}(\lambda) \sim \mathcal{N}\left(K_{n+1}K^{-1}\vec{f}(\lambda),\ k(\lambda_{n+1}, \lambda_{n+1}) - K_{n+1}K^{-1}K_{n+1}\right),$$

allowing a closed-form calculation (Rasmussen and Williams, 2006).

The calculation of the expected improvement for a given configuration is straight forward

$$\begin{aligned} EI(\lambda) &= \int_{-\infty}^{f'} \left(f' - f\right) \cdot p\left(f \mid D\right) \mathrm{d}f \\ &= (m(\lambda) - f') \cdot \phi\left(\frac{m(\lambda) - f'}{k(\lambda, \lambda)}\right) + \sigma(\lambda) \cdot \mathcal{N}\left(\frac{m(\lambda) - f'}{k(\lambda, \lambda)}\right) \end{aligned}$$

with $\mathcal{N}$ the standard normal probability density function and ϕ the standard normal cumulative distribution function.

A common drawback of Gaussian processes is the runtime complexity of $\mathcal{O}(n^3)$ due to the inversion of K (Rasmussen and Williams, 2006). However, as long as multi-fidelity methods (see Section 7) are not used, this is not relevant as evaluating a high number

of configurations is prohibitively expensive. A more relevant drawback for CASH is the missing native support of categorical input; yet, extensions for treating integer variables in Gaussian processes exist, e.g., (Garrido-Merchán and Hernández-Lobato, 2017).

Furthermore, the selection of an appropriate covariance function $k(\lambda, \lambda')$ is difficult for high dimensional input. To obtain good results, the hyperparameters of the Gaussian process have to be manually fine tuned requiring again expert knowledge or many trials.

4.3.2 Random Forests

Random forest regression (Breiman, 2001) is an ensemble method consisting of multiple regression trees (Breiman et al., 1984). Regression trees—or in general prediction trees as classification and regression basically behave the same—use recursive splitting of the training data to create groups of similar observations. This splitting can be visualized as a tree structure.

To build a prediction tree, a root element containing all training data is created. This root element is split into several child nodes by evaluating a single feature x. x is selected such that the purity of all child nodes—measured by an *impurity function*—becomes maximal. Common impurity functions are *information gain* from information theory or *gini impurity* (Raileanu and Stoffel, 2004). It is important to note that the domain of x is not relevant for the impurity function, allowing splitting of categorical and numerical features. This procedure is repeated for every child node until the impurity reaches a predefined threshold. These so-called leaf nodes contain an aggregated label, for example the average value for regression or the most common class for classification.

Besides the ability to natively handle categorical variables, decision trees are also fast to train and even faster on evaluating new data. However, decision trees are also prone to overfitting, especially when a tree is particularly deep (Breiman et al., 1984). Furthermore, the best split is only chosen based on local information. Even though every best local decision is made, the global performance may still be suboptimal. Building a globally optimal decision tree is an NP-hard problem (Breiman, 2001).

Both drawbacks can be limited by using random forests. Instead of training a single prediction tree, multiple trees are created. As the creation of a single tree is deterministic, the training data for each tree has to vary. By randomly dropping some observations as well as some features—referred to as *bootstrapping*—each prediction tree is trained on a unique data set. When predicting the label of a new observation, the result of each individual prediction tree is aggregated using *bagging* (Breiman, 2001). In the most basic case, bagging is implemented via averaging the results. The drawback of evaluating multiple prediction trees is negligible as the prediction runtime is limited by $\mathcal{O}(\log n)$ for a relatively balanced tree.

To compute the expected improvement for a given configuration λ, the mean and variance have to be known. The predictive mean $\mu(\lambda)$ is calculated by averaging the results of each individual regression tree; the predictive variance $\sigma^2(\lambda)$ is given by the sample variance

$$\sigma^2(\lambda) = \frac{1}{n-1} \sum_i \left(f_i(\lambda) - \mu(\lambda)\right)^2$$

of the results f_i of each regression tree i.

4.3.3 Tree-structured Parzen Estimator

In contrast to the previous surrogate models, a tree-structured Parzen estimator (TPE) (Bergstra et al., 2011) does not model the posterior $p(f \mid D)$ directly. Instead the likelihood $p(D \mid f)$ and marginal likelihood $p(D)$ are modeled. Based on a given quantile γ, all previous observations D are split into two groups using a calculated threshold f', such that $\gamma = p(f < f')$. Using f', $p(D \mid f)$ is constructed as

$$p(D \mid f) = \begin{cases} l(D) & \text{if } f < f' \\ g(D) & \text{if } f \geq f'. \end{cases} \tag{8}$$

Two different distributions for the hyperparameters are used: one for well performing configurations called $l(D)$ and one for bad performing configurations called $g(D)$. These two distributions are created using kernel density estimations (KDEs) (Parzen, 1961). $p(D)$ is defined as

$$\begin{aligned} p(D) &= \int_{\mathbb{R}} p(D \mid f) \cdot p(f) \, \mathrm{d}f \\ &= \gamma \, l(D) + (1 - \gamma) g(D). \end{aligned}$$

It is important to note that $p(f)$ is not required to construct $p(D \mid f)$ or $p(D)$.

Regarding the tree structure, TPE natively handles hierarchical search spaces by modeling each hyperparameter individually by two one-dimensional distributions. These distributions are connected hierarchically representing the dependencies between the hyperparameters resulting in a pseudo multidimensional distribution.

The complexity of constructing the model is dominated by finding the current threshold f'. By keeping a sorted list of all previous observations, the γ quantile can be calculated in $\mathcal{O}(n)$ (Bergstra et al., 2011).

Expected improvement is build based on the construction of $p(D \mid f)$ and $p(D)$. It is defined as

$$\begin{aligned} EI(\lambda) &= \int_{-\infty}^{f'} \left(f' - f\right) \cdot p\left(f \mid D\right) \mathrm{d}f \\ &= \int_{-\infty}^{f'} \left(f' - f\right) \cdot \frac{p\left(D \mid f\right) p\left(f\right)}{p\left(D\right)} \mathrm{d}f \\ &= \frac{\gamma \, f' \, l(\lambda) - l(\lambda) \int_{-\infty}^{f'} p(f) \, \mathrm{d}f}{\gamma \, l(\lambda) + (1 - \gamma) g(x)} \\ &\propto \left(\gamma + (1 - \gamma) \frac{g(\lambda)}{l(\lambda)}\right)^{-1}. \end{aligned}$$

Due to the proportionality in the last line, exact knowledge about $p(f)$ is not required.

4.4 Evolutionary Algorithms

An alternative to SMBO are evolutionary algorithms (Coello et al., 2007). Evolutionary algorithms are a collection of various population-based optimization algorithms inspired by biological evolution. All instances of evolutionary algorithms follow the same abstract procedure visualized in Figure 5. At first, an initial population is randomly created. The performance of each individual in the current generation is evaluated using an objective function and the best performing individuals are selected. Based on these parents a new generation is created. These three steps are repeated until termination.

In general, evolutionary algorithms are applicable to a wide variety of optimization problems as no assumptions about the objective function are necessary. Due to the somewhat random and unguided creation of new generations, evolutionary algorithms are not very efficient concerning the number of objective function evaluations (Cooper and Hinde, 2003) in comparison to Bayesian optimization. However, as all individuals of one generation are evaluated independently of the rest, this approach is especially well suited for distributing the computational load in a cluster.

The methodology of breeding a new generation depends on the specific algorithm instance. In the following two popular algorithm instances are presented.

4.4.1 Genetic Programming

The concept of genetic programming was already introduced in Section 3.2.1 with the focus on building variable shaped pipeline structures. The same principles can be applied for CASH with some limitations. Genetic programming works quite naturally for HPO of a fixed algorithm: the configuration space is discretized using the same mechanics as for grid search (see Section 4.1). Each individual represents the selected algorithm with the according hyperparameters, modeled as a configuration tree. Shrinking is implemented by removing hyperparameter nodes and therefore the default value for this hyperparameter is used. Insertion works complementary: a non-configured hyperparameter is set to a fixed value. Finally, point-mutation changes the value of a single hyperparameters. Crossover between identical algorithms is implemented by selecting two random subsets of the hyperparameters and merging them.

However, these concepts cannot be easily applied to algorithm selection: a crossover between a SVM and a decision tree cannot be modeled reasonably. Consequently, genetic programming should only be used for CASH in combination with ML pipeline structure search.

4.4.2 Particle Swarm Optimization

Another evolutionary algorithm instance is a particle swarm (Reynolds, 1987). Originally developed to simulate simple social behavior of individuals in a swarm, particle swarms can also be used as an optimizer (Kennedy and Eberhart, 1995).

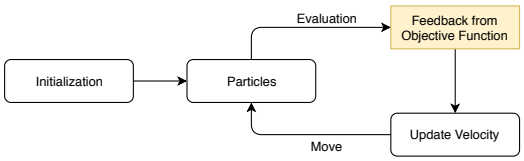


Figure 9: Schematic procedure of PSO. A set of particles with random position and velocity is created. Each particle is evaluated against the objective function and the velocity is updated. Finally, each particle moves to a new position based on its velocity.

Particle swarm optimization (PSO) uses a group of n search agents, called particles, organized in a swarm

$$\mathcal{S} = (S_1, S_2, \ldots, S_n).$$

Each particle is defined by a triple

$$S_i = \langle \vec{x}_i, \vec{v}_i, \vec{p}_i \rangle$$

with $\vec{x}_i$ the current position, $\vec{v}_i$ the current velocity of the particle and $\vec{p}_i$ the best position of the particle so far, for $i \in 1, \ldots, n$.

PSO starts by generating particles with a random position in the search space and a random velocity. Each particle calculates the performance of the current position and memorizes the position and score. Next, the speed for each particle is adjusted such that the particle moves to a better solution based on the previously visited positions and the globally best known position $\vec{p}$

$$\vec{v}_i \coloneqq \vec{v}_i + \mathcal{U}(0, \phi_1) \cdot (\vec{p}_i - \vec{x}_i) + \mathcal{U}(0, \phi_2) \cdot (\vec{p} - \vec{x}_i)$$

with $\mathcal{U}(0, \phi.)$ being continuous uniform distributions based on user defined acceleration constants ϕ_1, ϕ_2. Finally, each particle moves according to its individual velocity vector

$$\vec{x}_i \coloneqq \vec{x}_i + \vec{v}_i$$

and the procedure starts form the beginning. These three steps, visualized in Figure 9, are repeated for a fixed budget or until convergence (Parsopoulos, 2016).

Inherently, a particle's position and velocity are defined by continuous vectors $\vec{x}_i, \vec{v}_i \in \mathbb{R}^d$. Similar to Gaussian processes, all categorical and integer hyperparameters have to be mapped to continuous variables introducing a mapping error.

Even though the general procedures are quite similar, there are some differences between PSO and genetic programming: PSO does not handle the explicit merging of two solutions like genetic programming. Therefore, a particle has limited memory of all previously tested solutions. In consequence, PSO tends to converge faster to a local minimum in comparison to genetic programming (Parsopoulos, 2016).

4.5 Multi-Armed Bandit Learning

In Section 4.3 the general problem of exploitation versus exploration was shortly introduced. An extensively studied model for this problem is the *multi-armed bandit problem* (Robbins, 1952). A gambler has access to K different slot machines and each machine provides a reward based on an unknown distribution. The gambler is allowed to pull a fixed number of arms of any machine he wants to use. To maximize his reward, the gambler has to find the arm with the highest probability of reward as soon as possible and then keep using this arm.

Formally, the bandit problem is defined as a game between a player and an adversary. The game is parametrized by K actions, denoted by $1 \leq i \leq K$. Each action has a hidden underlying distribution providing a random reward each time the action is chosen. For $t = 1, 2, \ldots, T$ trials the following two steps are repeated (Auer et al., 1995):

1. The adversary selects a random reward $\vec{x}^{(t)} \in \mathbb{R}^K$ from the hidden distributions for each action based on the previously selected actions.

2. The player chooses an action i and receives the random reward $x_i^{(t)}$.

After each trial, the individual rewards for each action are aggregated in a cumulative bandit reward $z_i^{(T)}$. Therefore, let $\vec{x}_i = [x_i^{(1)}, \ldots, x_i^{(T)}]$ denote all rewards obtained for action i and $|\vec{x}_i|$ the cardinality of $\vec{x}_i$. Common choices for calculating a cumulative reward are averaging all rewards

$$z_i^{(T)} = \frac{1}{|\vec{x}_i|} \sum_{t=1}^{|\vec{x}_i|} x_i^{(t)}$$

or computing the velocity of improvements by

$$z_i^{(T)} = \frac{1}{|\vec{x}_i|} \sum_{t=2}^{|\vec{x}_i|} x_i^{(t)} - x_i^{(t-1)}$$

over a sorted list $\vec{x}_i$ (Swearingen et al., 2017).

The strategy to decide which action to choose next is called a *policy*. In the literature many different policies have been proposed, for example greedy and ϵ-greedy methods (Langford and Zhang, 2008) or the *upper confidence bound* (Auer, 2002).

Multi-armed bandit learning is limited to a finite number of bandits. This makes bandit learning especially useful for selecting values for categorical hyperparameters, e.g selecting an ML algorithm (Hoffman et al., 2014; Efimova et al., 2017; Gustafson, 2018; das Dôres et al., 2018). However, continuous hyperparameters cannot be modeled directly.

4.6 Gradient Descent

A very powerful optimization method is *gradient descent* (Bottou, 2010), an iterative minimization algorithm. Starting from a random point λ_0, the algorithm moves in the opposite direction of the largest gradient to select the next point λ_1 as

$$\lambda_{n+1} = \lambda_n - \gamma \cdot \nabla f(\lambda_n)$$

with $\gamma \in \mathbb{R}_+$ being a constant step size. This way, a monotonic sequence $f(\lambda_0) \geq f(\lambda_1) \geq \ldots \geq f(\lambda_n)$ converging to a local minimum is created. If the objective function is convex, the local minimum also represents the global minimum.

If f is differentiable and its closed-form representation is known, the gradient ∇f is computable. However, for CASH the closed-form representation of f is not known. Consequently, the gradient cannot be computed and gradient descent is not applicable. By assuming some properties of f—and therefore limiting the applicability of this approach to certain problems—gradient descent can still be used (Pedregosa, 2016):

- As gradient descent depends on the gradient of f, the objective function has to be differentiable. As consequence, f has also be continuous. This eliminates categorical and integer hyperparameters—even if encoded by index and relaxed to real-values—from the search space as they introduce non-continuities.
- The first and second derivative of f have to be Lipschitz continuous functions.
- The Hessian matrix has to be nonsingular to be invertible.

The gradient itself has not to be known in advance. Instead it can be approximated during the optimization (Pedregosa, 2016) or calculated explicitly through reversible learning (Bengio, 2000; Maclaurin et al., 2015).

The limitation to continuous hyperparameters renders these approaches not applicable for CASH as the algorithm selection is not possible. However, in combination with bandit learning (see Section 4.5) gradient descent is still useful. Even though gradient descent converges faster to local minima than the previously presented methods it is not analyzed in more details as the prerequisites limit it to very few problem instances.

5. Automatic Data Cleaning

Data cleaning is an important aspect of building an ML pipeline. The purpose of data cleaning is improving the quality of a data set by removing data errors. Common error classes are missing values in the input data, invalid values or broken links between entries of multiple data sets (Rahm and Do, 2000).

Most existing AutoML frameworks recognize the importance of data cleaning and include various data cleaning stages in the fitted ML pipeline, e.g., (Feurer et al., 2015a; Swearingen et al., 2017; Parry, 2019). However, these data cleaning steps are usually hard-coded and not generated based on some metric during an optimization period. These fixed data cleaning steps usually contain imputation of missing values, removing of samples with incorrect values, like infinity, or outliers and scaling features to a normalized range.

Sometimes, high requirements for specific data qualities are introduced by later stages in an ML pipeline. For example, consider categorical features in a classification task. If a SVM is used for classification, all categorical features have to be numerically encoded. In contrast, a random forest is able to handle categorical features. These additional requirements can be detected by analyzing a candidate pipeline and matching the prerequisites of every stage with meta-features of each feature in the data set (Gil et al., 2018).

All currently used data cleaning steps are domain-agnostic. Consequently, these general purpose steps are not able to catch domain specific errors. However, by incorporating

domain knowledge, the data quality can be heavily increased (Jeffery et al., 2006; Messaoud et al., 2011; Salvador et al., 2016). Using different representations of expert knowledge, like integrity constraints or first order logic, low quality data can be automatically detected and corrected (Raman and Hellerstein, 2001; Hellerstein, 2008; Chu et al., 2015, 2016). However, these potentials are not used by current AutoML frameworks as they aim to be completely data-agnostic to be applicable to a wide range of data sets. Consequently, advanced and domain specific data cleaning is conferred to the user.

6. Automatic Feature Engineering

Feature engineering is the process of generating and selecting features from a given data set for the subsequent modeling step. This step is crucial for the complete ML pipeline, as the overall model performance highly depends on the available features. By building good features, the performance of an ML pipeline can be increased many times over (Pyle, 1999). Yet, having too many unrelated feature increases the training time (curse of dimensionality) and could also lead to overfitting. The task of feature engineering is highly domain specific and very difficult to generalize. Even for a data scientist assessing the impact of a feature is difficult, as domain knowledge is necessary. Consequently, feature engineering is a mainly manual and time-consuming task driven by trial and error.

Feature selection is the easier part of feature engineering as the search space is very limited: each feature can either be included or excluded. By removing redundant or misleading features, an ML pipeline can be trained faster with a lower generalization error. Furthermore, the interpretability of the trained model is increased. By examining the features from an information theoretic or statistics viewpoint (Pudil et al., 1994; Yang and Pedersen, 1997; Dash and Liu, 1997; Guyon and Elisseeff, 2003), no domain knowledge is necessary for feature selection. Consequently, many different algorithms for feature selection exist—like univariate selection, variance threshold, feature importance or correlation matrices (Saeys et al., 2007)—and are already integrated in modern AutoML frameworks (Komer et al., 2014; Feurer et al., 2015a; Olson and Moore, 2016; Swearingen et al., 2017; Parry, 2019).

More interesting is the task of feature generation as the number of new features that can be derived from a given data set is unbounded. As being domain-agnostic is a core goal of AutoML, domain knowledge cannot be used for feature generation. Approaches to enhance automatic feature generation with domain knowledge, e.g., (Friedman and Markovitch, 2015; Smith et al., 2017), are therefore not considered. Still, some features—like dates or addresses—can be easily parsed without domain knowledge to create more meaningful features (Chen et al., 2018). However, for arbitrary numeric or categorical features such rule-based feature generation is not applicable requiring different measures.

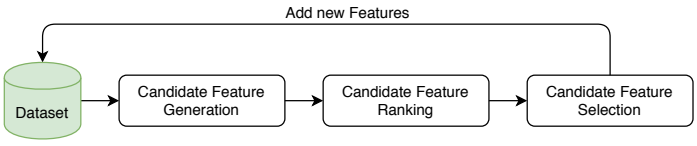


Figure 10: Iterative feature generation procedure.

Basically all automatic feature generation approaches follow the iterative scheme displayed in Figure 10. Based on an initial data set, a set of candidate features is generated and ranked. High ranking features are evaluated and potentially added to the data set. These three steps are repeated several times.

The only viable approach for candidate feature generation is *deep feature synthesis* (Kanter and Veeramachaneni, 2015). Let a set of features $F_{\text{init}} = \{f_1, \ldots, f_d\}$ and a set of operators be given:

Unary Unary operators transform a single feature, for example by discretizing or normalizing numerical features, applying rule-based expansions of dates or using unary mathematical operators like a logarithm.

Binary Binary operators combine two features, e.g., via basic arithmetic operations. A special type of binary operators are feature correlations. Using correlation tests and regression models, the correlation between two features can be expressed as a new feature (Kaul et al., 2017; Chen et al., 2018).

High-Order High-order operators are usually build-around the SQL *Group By* operator: all records are grouped by one feature and then aggregated via minimum, maximum, average or count.

Deep feature synthesis exhaustively applies all operators on all features creating an exponential set of new features F_{cand}. By repeating this procedure k times, deep features are obtained. This process is can be interpreted as a *transformation tree* T: the root node represents the original features; each edge applies one specific operator leading to a transformed feature set (Khurana et al., 2016; Lam et al., 2017).

Deep feature synthesis generates a great many features; with a high probability most of those features do not contain meaningful information. Consequently, the set of candidate features F_{cand} has to be filtered. A common strategy is treating feature ranking as a graph traversal problem: given a complete transformation tree T, the node with the minimal loss has to be found. A simple approach is a greedy ranking based on the candidate parents performance (Khurana et al., 2016). A more sophisticated approach is selecting nodes based on reinforcement learning. By combining the parent's performance with features of T like node depth or number of different operators leading to the selected node, a policy can be constructed (Khurana et al., 2018a). This way, a better trade-off between exploration and exploitation can be achieved.

A totally different approach is ranking candidate features based on meta-features. Meta-features describe the properties of a feature f_i, like the distribution of values in f_i. Based on the meta-features of a candidate feature, but also meta-features of its parents, the expected loss reduction after including this candidate can be predicted using a regression model (Katz et al., 2017; Nargesian et al., 2017). The regression model is created in an offline training phase.

Finally, candidate features are selected by their ranking and the best features are added to the data set. This procedure is repeated for k times.

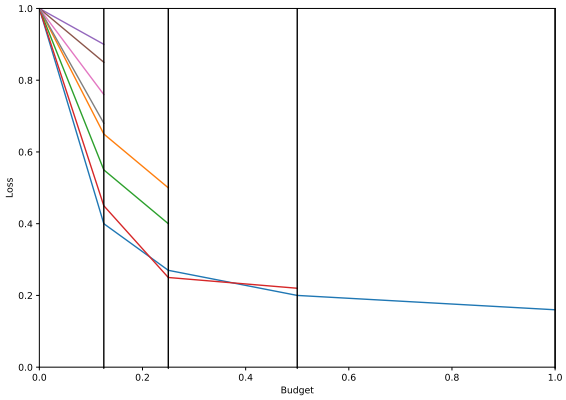


Figure 11: Schematic representation of SuccessiveHalving. At first, eight different configurations are tested on ⅛ of the total budget. The better performing halve is transferred to the next iteration with doubled budget. This procedure is repeated until only one configuration remains (Hutter et al., 2018).

7. Performance Improvements

In the previous sections various techniques for building an ML pipeline have been presented. In this section different performance improvements are introduced. These improvements cover multiple techniques to speed up the optimization procedure as well as improving the overall performance of the generated ML pipeline.

7.1 Multi-Fidelity Approximations

The major problem for AutoML and especially CASH procedures is the extremely high turnaround time. Depending on the used data set, fitting a single model can take several hours, in extreme cases even up to several days (Krizhevsky et al., 2012). Consequently, optimization progress is very slow. A common approach to circumvent this limitation is the usage of multi-fidelity approximations. Data scientist often use only a subset of the training data or a subset of the available features (Bottou, 2012). By testing a configuration on this training subset, bad performing configurations can be discarded very fast and only well performing configurations have to be tested on the complete training set. The methods presented in this section aim to mimic this manual procedure to make it applicable for fully automated ML.

A straight-forward approach to mimic expert behavior is choosing multiple random subsets of the training data for performance evaluation (Nickson et al., 2014). More sophisticated methods augment the black box optimization in Equation (3) by introducing an additional budget term

$$f : \mathbb{X}^d \times \mathbb{R} \to \mathbb{Y}.$$

The generic budget s, usually represented as a fraction $s \in [0, 1]$, can be freely selected by the optimization algorithm. s can be interpreted in multiple ways, e.g., fraction of the training data or maximum number of iterations for local optimization.

SuccessiveHalving (Jamieson and Talwalkar, 2015) solves the selection of s via bandit learning. The basic idea, visualized in Figure 11, is very simple: SuccessiveHalving randomly creates m configurations and tests each for the partial budget $s_0 = 1/m$. The better half is transferred to the next iteration allocating twice the budget $s_{t+1} = 2 \cdot s_t$ to evaluate each remaining configuration. This procedure is repeated until only one configuration remains. Consequently, well performing configurations receive exponentially more training time than bad performing ones. A crucial problem with SuccessiveHalving is the selection of m for a fixed budget: is it better to test many different configurations with a low budget or only a few configurations with a high budget?

hyperband (Li et al., 2016, 2018) answers this question by dynamically selecting an appropriate number of configurations. As shown in Algorithm 2, hyperband calculates the number of configurations and budget size based on some budget constraints $b_{\min}$ and $b_{\max}$. The budget constraints are normally defined by the underlying problem, for example the training sample size is limited by the number of available samples and the minimum number of samples required to train a sensible model. In a loop a descending sequence of configuration numbers m is calculated (lines 2–3). Based on these numbers candidate configurations are sampled (line 4) and passed to SuccessiveHalving (line 5). Consequently, no prior knowledge is required anymore for SuccessiveHalving.

Algorithm 2 hyperband

Input: b_{min}, b_{max}, Λ

1: $b_{max} \leftarrow \log\left\lfloor \frac{b_{max}}{b_{min}} \right\rfloor$
2: **for** $b \in \{b_{max}, b_{max} - 1, \ldots, 0\}$ **do**
3: $\quad m \leftarrow \text{DetermineBudget}(b)$
4: $\quad \vec{\lambda} \leftarrow \text{SampleConfigurations}(m, \Lambda)$
5: $\quad \text{SuccessiveHalving}(\vec{\lambda})$
6: **end for**

An alternative to SuccessiveHalving and hyperband is Fabolas (Klein et al., 2016). Instead of using deterministically calculated budgets s, Fabolas uses multi-objective optimization to reduce model loss and training time. Therefore, a Gaussian process is trained on the combined input (λ, s). Additionally the acquisition function is enhanced by entropy search (Hennig and Schuler, 2012). This allows predicting the performance of λ_i, tested with budget s_i, for the full budget $s = 1$. Fabolas follows the standard procedure for SMBO displayed in Algorithm 1 closely. Yet, an additional step to calculate the incumbent

$$\hat{\lambda}_i = \arg\max_{\lambda_j \in D} \; f(\lambda_j, 1)$$

for the next iteration is necessary.

7.2 Early Stopping

In contrast to using only a subset of the training data, several methods have been proposed to terminate the evaluation of probably bad performing configurations early. Many existing AutoML frameworks (see Section 8) incorporate k-fold cross-validation to limit the effects

of overfitting. A quite simple approximation is aborting the fitting after the first fold if the performance is significantly worse than the current incumbent (Hutter et al., 2011). Consequently, $k-1$ probably superfluous fitting procedures can be omitted.

A more elaborate approach is Freeze-Thaw Bayesian optimization (Swersky et al., 2014). This approach treats CASH as a nested optimization problem: in an outer loop, a solver tries to find the optimal hyperparameter configuration; in an inner loop, a different solver tries to find a model that minimizes the loss for the given hyperparameters. The basic idea of Freeze-Thaw Bayesian optimization is stopping the inner optimization loop early—called *freeze*—if it seems unlikely that it will be able to find a model with a small loss. Instead, another hyperparameter configuration is going to be tested. However, the state of the inner optimization loop is saved allowing picking up the optimization again—called *thaw*—if the chances for a small loss for this specific hyperparameters have increased. It is assumed that the improvements of the inner optimization loop can be modeled as an exponential decay. Based on the last improvements, the predicted improvement is updated. In each iteration, the configuration with the highest predicted performance is optimized.

In non-deterministic scenarios, configurations usually have to be evaluated on multiple problem instances to obtain reliable performance measures. Some of these problem instances may be very unfavorable leading to very long optimization periods. By evaluating multiple problem instances in parallel, a dynamic runtime threshold can be computed to abort long running instances while still preserving the theoretical quality guarantees (Weisz et al., 2018).

7.3 Scalability

As previously mentioned, fitting an ML pipeline is a time consuming and computational expensive task. A common strategy for solving a computational heavy problem is parallelization on multiple cores or within a cluster, e.g., (Buyya, 1999; Dean and Ghemawat, 2008). ML in total is an inherently sequential task that is hard to parallelize. However, some ML algorithms are easy to parallelize, for example fitting of multiple prediction trees in a random forest can be easily scaled. Furthermore, research exists to parallelize sequential ML algorithms, e.g., (Zeng et al., 2008). scikit-learn (Pedregosa et al., 2011) already implements many optimizations to distribute workload on multiple cores on a single machine. As AutoML normally has to fit many ML models, distributing different fitting instances in a cluster is an obvious idea.

Random search, grid search and evolutionary algorithms allow easy parallelization of single evaluations as pipeline instances are independent of each other. Gradient descent—in a stochastic variation—and bandit learning (Desautels et al., 2014) can be parallelized with some efforts (Zinkevich et al., 2010; Desautels et al., 2014). However, SMBO is—as the name already implies—a sequential procedure. Consequently, evaluating multiple configurations at once requires some adaptations. A possible solution is selecting the best n configurations instead of only the best configuration (Bergstra et al., 2011; Hutter et al., 2011). After evaluating all configurations, the surrogate model is updated and the next batch of configurations evaluated. Due to different evaluation runtimes, this *synchronous* approach does not utilize the available resources very efficiently. Alternatively, an uncompleted evaluation of a configuration can be modeled as the worst possible result (Kandasamy

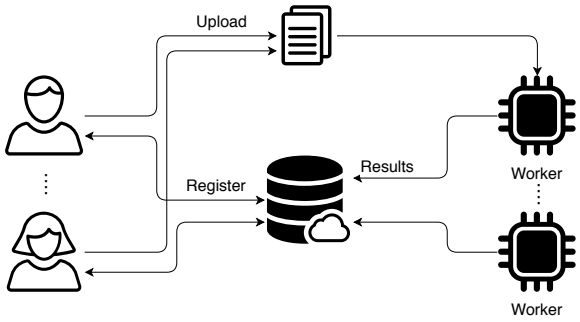


Figure 12: Components of an AutoML service (Swearingen et al., 2017).

et al., 2017). This way, new configurations can be sampled from an approximated posterior while preventing evaluating the same configuration twice. In addition, the different evaluations can be executed *asynchronously* without any synchronization leading to a better utilization of the available resources.

The scaling of AutoML tasks to a cluster also allows the introduction of AutoML services. Users can upload their data set and configuration space—called a *study*—to a persistent storage. Workers in a cluster test different configurations of a study until a budget is exhausted. This procedure is displayed in Figure 12. As a result, users can obtain optimized ML pipelines with minimal effort in a short timespan.

Various open-source designs for AutoML services have been proposed, e.g., (Sparks et al., 2015; Chan, 2017; Swearingen et al., 2017; Koch et al., 2018), but also several commercial solutions exist, e.g., (Golovin et al., 2017; Clouder, 2018; H2O.ai, 2018). Some commercial solutions also focus on providing ML without the need to write own code, enabling domain expert without programming skills to create optimized ML workflows (USU Software AG, 2018; Baidu, 2018; RapidMiner, 2018).

7.4 Ensemble Learning

A well-known concept in ML is ensemble learning. Ensemble methods combine multiple ML models to create predictions. Depending on the diversity of the combined models, the overall accuracy of the predictions can be significantly increased. The cost of evaluating multiple ML models is often neglectable considering the performance improvements.

During the search for a well performing ML pipeline, AutoML frameworks create a large number of different pipelines. Instead of only yielding the best performing configuration, the set of best performing configurations can be used to create an ensemble (Lacoste et al., 2014; Feurer et al., 2015a; Wistuba et al., 2017). As a consequence, AutoML procedures are capable of yielding a better overall performance.

An interesting approach for ensemble learning is *stacking* (Wolpert, 1992). A stacked ML pipeline is generated in multiple layers, each layer being a *normal* ML pipeline. The predicted output of each previous layer is appended as a new feature to the training data of subsequent layers. This way, later layers have the chance to correct wrong predictions of earlier layers. In the end, all predictions are combined via bagging (Kaul et al., 2017; Chen et al., 2018).

Automatic feature engineering often creates several different candidate data sets (Khurana et al., 2016; Katz et al., 2017; Nargesian et al., 2017). By using multiple data sets, various ML pipelines can be constructed (Khurana et al., 2018b). This ensemble of pipelines can significantly boost the overall performance.

7.5 Meta-Learning

Given a new unknown ML task, AutoML methods usually start from scratch to build an ML pipeline. However, a human data scientist does not always start all over again but learns from previous tasks. Meta-learning is the science of learning how ML algorithms learn. Based on the observation of various configurations on previous ML tasks, meta-learning builds a model to construct promising configurations for a new unknown ML task leading to faster convergence with less trial and error (Vanschoren, 2018a).

Meta-learning can be used in multiple stages of automatically building an ML pipeline to increase the efficiency:

Search Space Refinements All presented CASH methods require an underlying search space definition. Often these search spaces are chosen arbitrary without any validation leading to either bloated spaces or spaces missing well-performing regions. In both cases the AutoML procedure is unable to find optimal results. Meta-learning can be used to asses the importance of single hyperparameters allowing to remove unimportant hyperparameters from the configuration space (Hutter et al., 2014; van Rijn and Hutter, 2018; Probst et al., 2019).

Filtering of Candidate Configurations Many AutoML procedures generate multiple candidate configurations usually selecting the configuration with the highest expected improvement. Meta-learning can be used to filter empirically bad performing candidate configurations (Alia and Smith-Miles, 2006; Wistuba et al., 2015a). Consequently, the risk of superfluous configuration evaluations is minimized.

Warm-Starting Basically all presented methods have an initialization phase where random configurations are selected. However, the same methods as for filtering candidate configurations can be applied to initialization. Warm-starting can be used for many aspects of AutoML, yet most research focuses on model selection and tuning (De Miranda et al., 2012; Reif et al., 2012; Feurer et al., 2015a,b; Wistuba et al., 2015b; Lindauer and Hutter, 2018).

Pipeline Structure Meta-learning is also applicable for pipeline structure search. Using information which preprocessor and model combination perform well together, potentially better performing pipelines can obtain a higher ranking (Schoenfeld et al., 2018).

To actually apply meta-learning for any of these areas, a set of prior evaluations

$$\mathbf{P} = \bigcup_{t_j \in T, \lambda_i \in \Lambda} \pi(t_j, \lambda_i)$$

with T being the set of all known ML tasks, has to be given. Each record in this set contains the ML task t_j, selected configuration λ_i and calculated performance $\pi(t_j, \lambda_i)$. Given a new task t_{new}, a meta-learner L is trained on $\mathbf{P}$ to recommend configurations $\vec{\lambda}_{\text{new}}$.

A simple, task-independent approach for ranking configurations is sorting $\mathbf{P}$ by the performance. Configurations λ_i with a higher performance are more favorable (Vanschoren, 2018b). For configurations with similar performance, the training time can be used to prefer faster configurations (van Rijn et al., 2015). Yet, ignoring the task can lead to useless recommendations, for example a configuration well performing for a regression task may not be applicable to a classification problem.

A task $t_j \in T$ can be described by a vector $\vec{m}(t_j)$ of meta-features. Meta-features describe the training data set, e.g., number of instances or features, distribution of and correlation between features or measures from information theory. Using the meta-features of a new task $m(t_{\text{new}})$, a subset of $\mathbf{P}' \subset \mathbf{P}$ with similar tasks can be obtained. $\mathbf{P}'$ is than used similarly to task-independent meta-learning (Vanschoren, 2018b).

8. Existing Frameworks

This section provides an introduction to the most popular open-source AutoML frameworks. At first, implementations of CASH algorithms are presented and analyzed in Section 8.1. Next, frameworks for creating complete ML pipelines are discussed in Section 8.2. In this section all presented implementations are discussed qualitatively, experimental evaluation is provided in Section 9.

8.1 CASH Algorithms

At first popular implementations of methods for algorithm selection and HPO are discussed. The mathematical foundation for all discussed implementations was provided in Section 4 and Section 7. A summary including the most important features is available in Table 1.

8.1.1 Grid Search

Grid search is the classic approach for HPO with many different implementations. For the experiments the existing GridSearchCV implementation from scikit-learn (Pedregosa et al., 2011) is utilized. Besides a parallelization to evaluate several configuration instances at the same time on a single machine, the scikit-learn implementation does not provide any performance improvements. To ensure fair results, a mechanism for stopping the optimization after a fixed number of iterations has been added. For each configuration instance, the performance is calculated using cross-validation.

By design, GridSearchCV is limited to HPO for a fixed algorithm. To extend this implementation for algorithm selection, a distinct GridSearchCV instance is created for each available ML algorithm. This allows sequential evaluation of all available ML algorithms while also reducing the search space significantly by eliminating redundant configurations, for example trying different number of neighbors for k nearest neighbors while using a SVM for classification. When all grid search instances have finished, the best result of all instances is returned.

Algorithm	Solver	Λ	Parallel	Timeout
Grid Search	Grid Search	no	Local	no
Random Search	Random Search	no	Local	no
Spearmint	SMBO with Gaussian process	no	Cluster	no
RoBO	SMBO with various models	no	no	no
BTB	Bandit learning and Gaussian process	yes	no	no
hyperopt	SMBO with TPE	yes	Cluster	no
SMAC	SMBO with random forest	yes	Local	yes
BOHB	Bandit learning and TPE	yes	Cluster	yes
Optunity	Particle Swarm Optimization	yes	Local	no

Table 1: Comparison of different CASH algorithms. Reported are the used solver, whether the search space structure is considered (Λ), if parallelization is implemented and whether a timeout for a single evaluation exists.

8.1.2 Random Search

The other classic approach for HPO is random search. This algorithm also has many different implementations, but again the scikit-learn (Pedregosa et al., 2011) implementation RandomizedSearchCV is used. RandomizedSearchCV tests a fixed number of random configurations in parallel on a single machine. For each tested configuration, the performance is calculated using cross-validation.

Similar to GridSearchCV, RandomizedSearchCV is also designed to optimize only a single estimator. Therefore, the ability to also select a random algorithm has been added. The RandomizedSearchCV code is wrapped to first select an algorithm and the according configuration space $\Lambda^{(i)}$ and then passed to the scikit-learn implementation.

8.1.3 Spearmint

Spearmint (Snoek et al., 2012) uses SMBO with a Gaussian process as a surrogate model for proposing configurations. As mentioned in Section 4.3.1, the selection of the mean function m and covariance function k are crucial for the regression performance. Spearmint uses a constant mean function $m(\lambda) = \vec{c}$ and an adjusted Matérn kernel. It is important to note, that this configuration contains meta-hyperparameters itself that are not subject to the CASH optimization: the constant mean $\vec{c}$, the length scales $\theta_{1:d}$ of the kernel and the observation noise ν. In combination with the expected improvement in Equation (5) as the acquisition function, these meta-hyperparameters are optimized via the marginalization of the *integrated expected improvement*

$$\hat{a}(\lambda, D) = \int \mathbb{E}\left[\max\left(0, f' - f(\lambda)\right)\right] \cdot p\left(\theta, \nu, \vec{c} \mid D_{1:n}\right) \mathrm{d}\theta.$$

Using a Monte Carlo estimation (Rubinstein and Kroese, 2008), the integrated expected improvement is efficiently acquired by slice sampling (Snoek et al., 2012).

A major limitation of Spearmint is the missing support of categorical hyperparameters. In combination with the missing support for conditional hyperparameters Spearmint is rather unsuited for CASH as many unnecessary fitting procedures with inactive parameters would be executed. This behavior can be compensated to some extend by using a dedicated Spearmint instance for each available estimator. The available iterations are uniformly distributed to all ML algorithms. Yet, this approach does not allow the sharing of knowledge between different estimators.

8.1.4 RoBO

RoBO (Klein et al., 2017) is a generic framework for general purpose Bayesian optimization. It supports many standard surrogate models like Gaussian processes or random forests; yet, also uncommon models like Bayesian neural networks (Springenberg et al., 2016). As RoBO is not specialized on CASH, a structured search space or categorical parameters are not supported. RoBO does not support any performance improvements except Fabolas.

8.1.5 BTB

A major limitation of Gaussian processes is the missing support for categorical variables. BTB (Gustafson, 2018) circumvents this limitation by multi-armed bandit learning. At first, a grid over all combinations of all categorical hyperparameters is created. Each point in this grid—referred to as a *hyperpartition*—is treated as a bandit with unknown reward. To propose a new candidate configuration, a hyperpartition is selected via multi-armed bandit learning. With the selection of one hyperpartition all categorical hyperparameters are fixed. Next, the remaining continuous hyperparameters are selected using Bayesian optimization with Gaussian processes similar to Spearmint. It is important to note that each hyperpartition uses a dedicated Gaussian process. The obtained score is used to train the Gaussian process and is treated as a reward for the hyperpartition. These two steps are repeated for a fixed number of iterations.

BTB provides multiple policies for selecting a hyperpartition as well as acquisition functions for Gaussian processes. In the context of this work, BTB selects hyperpartitions using the upper confidence bound as a policy. The acquisition function samples random configurations and orders them by their expected improvement (see Equation 5).

8.1.6 hyperopt

hyperopt (Bergstra et al., 2011) is a CASH solver based on SMBO. As surrogate models, TPEs are used. Instead of using just a single surrogate model, multiple instances are used to model hierarchical hyperparameters. However, these dependencies have to be explicitly stated leading to a redundant configuration space definition. The number of iterations is only limited in number and not in elapsed time.

To maximize the expected improvement in Equation (6), a large number of configurations is drawn from $l(\lambda)$ in Equation (8). Each candidate configuration is evaluated regarding $\frac{l(\lambda)}{g(\lambda)}$, corresponding to the intuitive assumption that a new configuration should have a high probability regarding $l(\lambda)$ and low probability regarding $g(\lambda)$ in Equation (8).

HYPEROPT can be easily parallelized. As the new candidate configurations are generated based on a distribution, the impact of a single observation is limited. Therefore, recently proposed configurations are simply ignored until their performance is evaluated. Even though the optimization becomes less efficient as candidates are generated with incomplete knowledge, the total wall clock time is still significantly reduced (Bergstra et al., 2011).

8.1.7 SMAC

SMAC (Hutter et al., 2011) is yet another solver for configuration selection based on SMBO. It was the first framework explicitly supporting categorical variables, making it especially suited for CASH. After an initialization with the default—or random if no default exists—configuration, the SMBO loop is repeated for a fixed number of iterations or fixed time budget. The performance of all previous configuration runs is modeled using random forest regression. The random forest contains ten regression trees that are trained via bootstrapping and the results are averaged. For each tree, the hyperparameters are left at their default value. The selection of these meta-hyperparameters is not further motivated. Candidate configurations are generated via local search around the so far tested configurations. Therefore, the current EI for all previous configurations is computed. Using the configurations with the highest EI, new configurations are created by changing a random hyperparameter to a proximate value. This procedure is repeated $n \gg 1000$ times. Additionally, new configurations are randomly sampled from the complete configuration space.

After generating many candidate configurations, they are tested. For a deterministic environment, the candidate configurations are sequentially tested against the incumbent and replace it when the performance is better. This procedure is stopped when a fixed time bound is exceeded or all configurations are tested. For non-deterministic scenarios or environments with multiple problem instances, the racing procedure is a bit more complex. The incumbent is reevaluated on new problem instances to obtain a higher confidence of its general performance. Furthermore, a challenger is repeatedly tested on growing random subsets of the problem instances used to evaluate the incumbent. If the average performance of the challenger is better than the incumbent on the current problem subset it becomes the new incumbent and the next challenger is tested. During the racing period multiple new configurations are tested and added to the random forest regression model.

A very interesting feature of SMAC is the build-in support to terminate configuration evaluations after a fixed timespan. This way, very unfavorable configurations are discarded quickly without slowing the complete optimization down. Furthermore, SMAC is fully parallelized to test multiple configurations at once.

8.1.8 BOHB

BOHB (Falkner et al., 2018) is a composed solver for the CASH problem. It is a combination of Bayesian optimization and HYPERBAND (Li et al., 2018). A limitation of HYPERBAND is the random generation of the tested configurations. BOHB replaces this random selection by a SMBO procedure. All function evaluations are stored in and modeled by a TPE. New configurations are drawn from $l(\lambda)$ in Equation (8) with all KDE bandwidths multiplied by a factor b_w to encourage exploration. Additionally, a constant fraction ρ of the

Framework	CASH Solver	Structure	Ensem.	Meta	Parallel	Timeout
ML-Plan	Grid Search	Variable	no	no	Local	no
TPOT	Genetic Prog.	Variable	no	no	Local	yes
hyperopt-sklearn	hyperopt	Fixed	no	no	no	yes
auto-sklearn	SMAC	Fixed	yes	yes	Cluster	yes
ATM	BTB	Fixed	no	no	Cluster	no
auto_ml	Grid Search	Fixed	yes	no	Local	no

Table 2: Comparison of different AutoML frameworks. Reported are the used CASH solver and pipeline structure. Furthermore it is listed whether ensemble learning (Ensem.), meta learning (Meta), parallel evaluation of pipelines or a timeout for a single evaluation are supported.

candidate configurations is sampled at random to comply with the theoretical guarantees of hyperband (Li et al., 2018).

For each function evaluation, BOHB passes the current budget and a configuration instance to the objective function. The interpretation of the budget is conferred to the user, meaning it can represent basically anything, e.g., the fraction of training data to use, available runtime or number of iterations.

8.1.9 Optunity

Optunity (Claesen et al., 2014) is a generic framework for CASH with a set of different solvers. In the context of this paper, only the PSO solver is used. Optunity supports a structured configuration space similar to hyperopt. All dependencies in the configuration space have to be explicitly stated leading to a redundant representation. Categorical hyperparameters are transformed to integer hyperparameters (by indexing), integer hyperparameters are treated as continuous hyperparameters. Before evaluating the objective function for a given configuration, all transformations are reversed by rounding and selecting a categorical value based on the index. Optunity limits the number of total objective function evaluations. Based on a heuristic, a suited number of particles and generations is selected for a given number of evaluations.

8.2 AutoML Frameworks

This section presents the most popular frameworks for AutoML. All presented frameworks are capable of building a complete ML pipeline based on the methods provided in Sections 3, 5, and 6. For algorithm selection and HPO, implementations from Section 8.1 are used. A summary is available in Table 2.

8.2.1 ML-Plan

ML-Plan (Mohr et al., 2018; Wever et al., 2018) is a framework for building arbitrary shaped ML pipelines based on HTNs. The basic problem of HTNs is assigning a score to

each node to find the best node in the graph. If the node only contains ML primitives, the pipeline can simply be tested by fitting it. However, if the selected pipeline is still incomplete, e.g., it contains a generic preprocessor step with no assigned algorithm, this is not possible. ML-Plan solves this problem by random path completion. Each non-terminal task is expanded in random ways until k complete pipelines are obtained. These k pipelines are then evaluated and the average score is assigned to the node. Finally, the resulting graph is scanned using a greedy breadth-first search.

As the pipeline structure is flexible, ML-Plan is prone for overfitting. To compensate this risk, the evaluation is split into two phases: First, the actual search is performed. Then, the candidate's performance is evaluated using a hold-out data set. ML-Plan currently only supports HPO via grid search. Furthermore, ML-Plan merely implements a parallel evaluation of pipelines but not other advanced performance improvements.

8.2.2 TPOT

TPOT (Olson and Moore, 2016; Olson et al., 2016b) is a framework for building and tuning arbitrary classification and regression pipelines. It uses genetic programming to construct flexible pipelines and to select an algorithm in each pipeline stage. Regarding HPO, TPOT can only handle categorical parameters; similar to grid search all continuous hyperparameters have to be discretized. In contrast to grid search, TPOT does not exhaustively test all different combinations but uses again genetic programming to fine-tune an algorithm.

TPOT's ability to create arbitrary complex pipelines makes it very prone for overfitting. To compensate this, TPOT optimizes a combination of high prediction accuracy and low pipeline complexity. Therefore, pipelines are selected from a Pareto front using a multi-objective selection strategy. The evaluation of the performance of all individuals of a single generation is parallelized to speed up optimization. In the end, TPOT returns the single best performing pipeline.

Genetic programming does not impose any constraints on the reproduction step leading to arbitrary shaped pipelines. However, in reality dependencies between different pipeline stages and constraints on the complete pipeline exist. For example TPOT could create a pipeline for a classification task without any classification algorithm (Olson et al., 2016a). To prevent such defective pipelines, RECIPE (de Sá et al., 2017) has been proposed. RECIPE limits the diversity of generated pipelines by enforcing conformity to a grammar. This way reasonable but still flexible pipelines can be created.

8.2.3 hyperopt-sklearn

hyperopt-sklearn (Komer et al., 2014) is a framework for fitting classification and regression pipelines. The pipeline shape is fixed to exactly one preprocessor and one classification or regression algorithm; all algorithms are taken from scikit-learn. Those two algorithms are selected and configured via hyperopt. In general, hyperopt-sklearn only provides a thin wrapper around hyperopt by introducing the fixed pipeline shape and adding a configuration space definition for each implemented algorithm. Besides the addition of a time budget per evaluation, no other performance improvements are implemented. To limit the effects of overfitting, cross-validation is used to evaluate the performance of a

single configuration. HYPEROPT-SKLEARN stops the optimization after a fixed number of iterations.

8.2.4 AUTO-SKLEARN

AUTO-SKLEARN (Feurer et al., 2015a, 2018) is a tool for building classification and regression pipelines. The pipelines all have a fixed structure: at first, a fixed set of data cleaning steps—including categorical encoding, imputation, removing variables with low variance and scaling—is executed. Next, an optional preprocessing and mandatory modeling algorithm are selected and tuned via SMAC. As the name already implies, AUTO-SKLEARN uses SCIKIT-LEARN for all ML algorithms.

In contrast to the other AutoML frameworks presented in this section, AUTO-SKLEARN does incorporate many different performance improvements. Testing pipeline candidates is improved via parallelization on a single computer or in a cluster. Additionally, each evaluation is limited by a time budget. AUTO-SKLEARN uses meta-learning to initialize the optimization procedure. This meta-learning is fueled by an extensive evaluation of different pipelines on 140 distinct data sets. The meta-learning foundation is not updated when new pipelines and data sets are evaluated. Additionally, AUTO-SKLEARN implements ensemble learning. Instead of only returning the best performing pipeline, an ensemble of the c best pipelines is created.

8.2.5 ATM

ATM (Swearingen et al., 2017) is a collaborative service to build optimized ML pipelines. This framework has a strong emphasis on parallelization allowing the distribution of single evaluations in a cluster. Currently, ATM uses a fixed pipeline structure with fixed data cleaning steps, one tunable preprocessing step followed by a tunable classification algorithm.[1] All tunable algorithms are based on SCIKIT-LEARN. Even though ATM supports different CASH algorithms, currently only BTB is available. To limit the effects of overfitting, cross-validation is used during the evaluation of a pipeline. Additional performance improvements are not implemented. ATM stops the optimization after either a fixed number of iterations or after exhausting a given time budget.

An interesting feature of ATM is the so-called MODELHUB. This central database stores information about data sets, tested configurations and their performances. By combining the performance evaluations with, currently not stored, meta-features of the data sets, a valuable foundation for meta-learning could be created. This catalog of examples could grow with every evaluated configuration enabling a continuously improving meta-learning.

8.2.6 AUTO_ML

AUTO_ML (Parry, 2019) is a AutoML framework specialized on natural language processing. Yet, it can also be used for generic classification and regression problems. AUTO_ML uses a fixed pipeline structure with fixed data cleaning, scaling and feature selection steps followed by a modeling stage. If applicable, also a feature engineering step is added to extract numerous features from textual input data. All preprocessing stages can be individually

1. Regression algorithms are currently not implemented.

turned on or off, but neither the order of the distinct stages can be altered nor new preprocessing steps are added during the optimization. Grid search and genetic programming are supported as CASH solvers. It is not possible to adjust the grid size or number of individuals and generations, consequently it is not possible to influence the optimization duration. However, a parallelization on a single machine is supported.

Besides the possibility to tune scikit-learn estimators, auto_ml provides interfaces to other popular ML libraries like TensorFlow (Abadi et al., 2016) or XGBoost (Chen and Guestrin, 2016). To improve the overall performance, auto_ml provides the possibility to train an ensemble of algorithms.

9. Experiments

In the previous sections several AutoML techniques for various subproblems of building an ML pipeline have been introduced and compared qualitatively. This section provides empirical evaluation of different CASH and pipeline building algorithms.

A major problem of testing CASH—and therefore also pipeline building—procedures is the long turnaround time: to actually evaluate the performance of a selected configuration, a model has to be trained and validated. Depending on the used data set, this procedure can take several minutes to hours preventing rapid progress (Krizhevsky et al., 2012). The rest of this section introduces various techniques to speed up the evaluation process and applies them to the algorithm presented in the previous sections. Furthermore, the comparability of the results is discussed.

9.1 Comparability of Results

In general, a reliable and fair comparison of different AutoML algorithms and frameworks is quite difficult due to different preconditions. Starting from incompatible interfaces, for example stopping the optimization after a fixed number of iterations or after a fixed timespan, to implementation details like refitting a model on the complete data set after cross-validation can heavily skew the performance comparison. Moreover, the scientific works that propose the algorithms often use different data sets for benchmarking purposes. Using agreed-on data sets with standardized search spaces for benchmarking, like it is done in other fields of research, e.g., (Geiger et al., 2012), would increase the comparability.

To solve some of these problems, the ChaLearn AutoML challenge (Guyon et al., 2015, 2016, 2018) has been introduced. The ChaLearn AutoML challenge is an online competition for AutoML[2] established in 2015. The challenge focuses on solving supervised learning tasks, namely classification and regression, using data sets from a wide range of domains without any human interaction. The challenge is designed such that participants upload AutoML code that is going to be evaluated on a task. A task contains a training and validation data set, both unknown to the participant. Given a fixed timespan on standardized hardware, the submitted code trains a model and the performance is measured using the validation data set and a fixed loss function. The tasks are chosen such that the underlying data sets cover a wide variety of complications, e.g., skewed data distribu-

2. Available at `http://automl.chalearn.org/`.

tions, imbalanced training data, sparse representations, missing values, categorical input and irrelevant features.

The ChaLearn AutoML challenge provides a good foundation for a fair and reproducible comparison of state-of-the-art AutoML frameworks. However, its focus on a competition between various teams makes this challenge unsuited for initial development of new algorithm. As the challenge is organized in rounds running for a limited timespan, it is not possible to use the ChaLearn AutoML challenge at any time. The black-box evaluation and missing knowledge of the used data sets make reproducing and debugging failing optimization runs impossible. Even though the competitive concept of this challenge can boost the overall progress of AutoML, additional measures are necessary for daily usage.

HPOlib (Eggensperger et al., 2013) aims to provide standardized data sets for the evaluation of CASH algorithms. Therefore, benchmarks using synthetic objective functions (see Section 9.2) and real data sets (see Section 9.4) have been defined. Each benchmark defines an objective function, a training and validation data set along with a configuration space. This way the benchmark data set is decoupled from the algorithm under development and can be reused by other researchers leading to more comparable evaluations. Where possible, benchmarks from HPOlib have been used for the empirical evaluations in this paper.

9.2 Synthetic Test Functions

A common strategy applied for many years is using synthetic test functions. Instead of optimizing the loss function of a real data set, a synthetic loss function is applied. These synthetic loss functions have an *interesting* response surface with, for example, many local minima or a high variance in the absolute gradient.

An often used function is the Branin function (Dixon and Szego, 1978). Defined over a two dimension space with $x_1 \in [-5, 10]$ and $x_2 \in [0, 15]$, the Branin function can be analytically computed as

$$f(\vec{x}) = \left(x_2 - \frac{5.1x_1}{4\pi^2} + \frac{5x_1}{\pi} - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10. \tag{9}$$

The corresponding response surface is displayed in Figure 13. In general, the global optimum for a synthetic test function can be analytically computed. In the case of the Branin function, the global minima are located at $\vec{x}^* = (-\pi, 12.275)$, $(\pi, 2.275)$ and $(9.42478, 2.475)$ with $f(\vec{x}^*) = 0.397887$.

Due to the closed-form representation, the synthetic loss for a given configuration can be computed in constant time. However, synthetic loss functions can only be used to simulate a single response surface of real-valued parameters. Consequently, these functions are only suited to simulate HPO without algorithm selection. The circumvention of real data also prevents the evaluation of data cleaning and feature engineering steps.

All CASH algorithms from Section 8 are tested on various synthetic test functions. All experiments were conducted on Intel Xeon E5 processors with 8 cores and 30 GB memory. Grid search and random search are used as base line algorithms. To guarantee fair results, all implementations are fixed to 250 evaluations of the test function. A limitation by runtime is not reasonable for synthetic test functions as a single evaluation requires less than 0.1 ms.

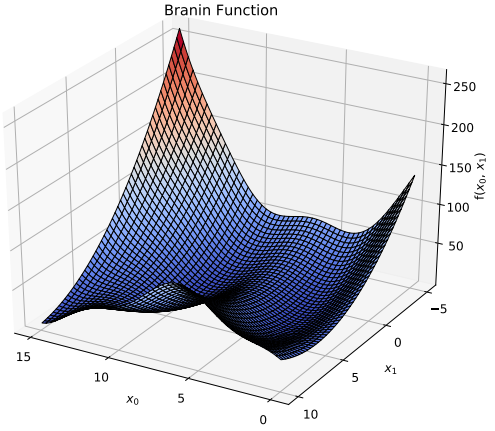


Figure 13: Response surface of the Branin function.

Each algorithm capable of parallel execution (see Table 1) was executed in eight parallel jobs on a single computer. Each experiment was repeated ten times with different seeds to account for non-deterministic procedures.

Grid search was configured to dynamically chose the number of points per dimension as $n = \lceil \sqrt[d]{250} \rceil$ to span the complete search space. Additionally, computations were stopped after 250 iterations to compensate for rounding errors. The budget estimate of BOHB is ignored as the actual test function evaluation cannot be accelerated. Consequently, BOHB basically behaves like HYPEROPT in combination with random search and some superfluous overhead to determine the budget. As the search space only contains continuous parameters, BTB creates just one hyperpartition behaving therefore like a normal SMBO algorithm with a Gaussian process as surrogate model. The available implementation of Spearmint is not compatible with Python 3. Therefore, the functionally identical implementation of RoBO (Klein et al., 2017) was used.[3]

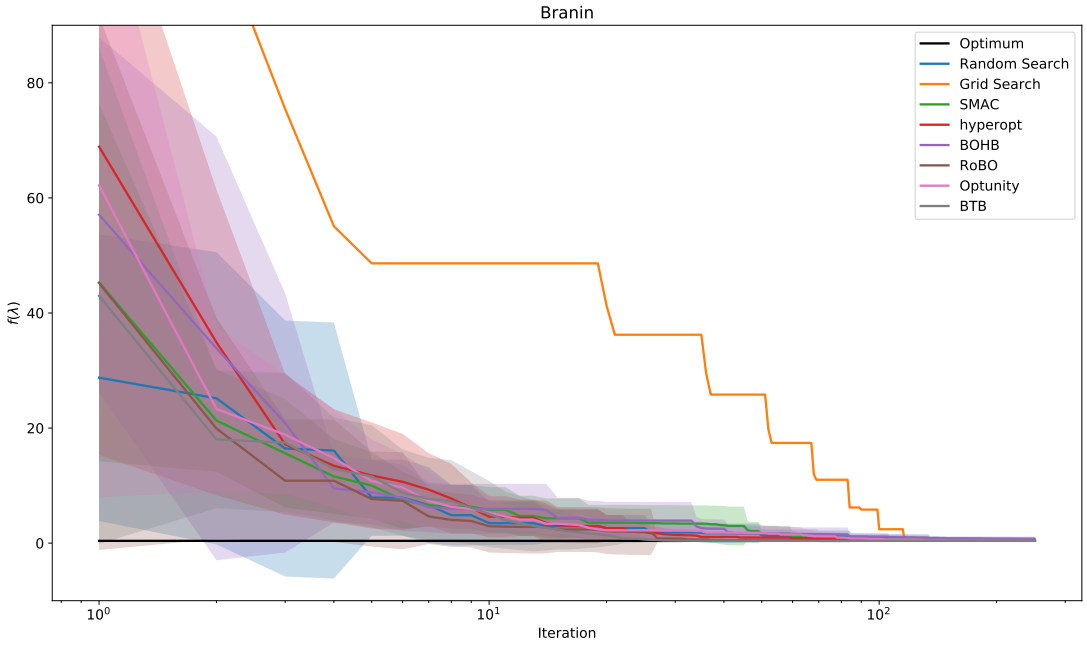
In the following, the results of two different synthetic test functions are discussed in detail. Further results of synthetic benchmarks are available in Appendix A. Figure 14 shows the average performance and standard deviation of the current incumbent in each iteration. Table 3 contains the performance of each algorithm after the completed optimization. The performance is defined as the minimal absolute distance

$$\min_{\lambda_i \in D} |f(\lambda_i) - f(\lambda^*)|$$

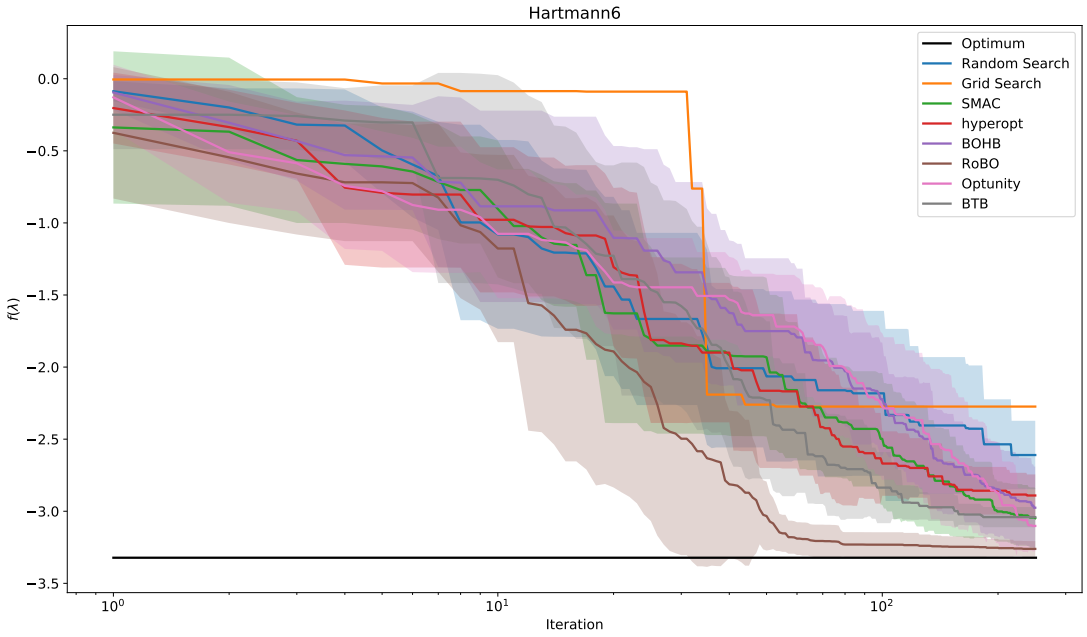
between the considered configurations λ_i and the global optimum λ^*.

In case of the two-dimensional Branin function (Dixon and Szego, 1978) all algorithms are consistently able to converge to the global minimum with a very small ϵ-offset within roughly 100 iterations as can be seen in Figure 14a. Grid search performs significantly worse over the first 100 iterations than all other algorithms but still converges to the global minimum later. All guided search strategies have very similar mean performances and standard

3. Unique features of SPEARMINT like multi-task optimization are not relevant for the conducted benchmarks.



(a) Branin function.



(b) Hartmann6 function.

Figure 14: Minimization progress of various CASH algorithms on two synthetic test functions. Displayed is the mean performance of the incumbent in each iteration. Additionally, the standard deviation over ten trials is shown.

Algorithm	Levy	Branin	Hartmann6	Rosenbrock10	Camelback
Grid Search	**0.00 ± 0.00**	0.25 ± 0.00	1.05 ± 0.00	09.00 ± 00.00	94.44 ± 00.00
Random Search	**0.00 ± 0.00**	0.27 ± 0.29	0.71 ± 0.24	46.10 ± 10.65	46.61 ± 30.39
RoBO	**0.00 ± 0.00**	**0.00 ± 0.00**	**0.06 ± 0.05**	**04.73 ± 02.02**	02.87 ± 06.17
BTB	0.18 ± 0.36	**0.00 ± 0.00**	0.28 ± 0.07	19.17 ± 03.99	07.75 ± 08.35
hyperopt	**0.00 ± 0.00**	0.06 ± 0.05	0.43 ± 0.15	24.01 ± 07.05	06.84 ± 06.04
SMAC	**0.00 ± 0.00**	0.10 ± 0.13	0.27 ± 0.21	36.75 ± 10.08	23.43 ± 27.29
BOHB	0.02 ± 0.03	0.36 ± 0.38	0.34 ± 0.29	34.54 ± 09.50	36.38 ± 39.86
Optunity	**0.00 ± 0.00**	0.03 ± 0.03	0.22 ± 0.18	35.66 ± 07.59	**01.75 ± 01.70**

Table 3: Results of all tested CASH solvers after 100 iterations. For each synthetic benchmark the mean performance and standard deviation over 10 trials is reported. Bold face represents the best mean value for each benchmark.

deviations. Random search performs basically identical to the guided search strategies. This implies that a two dimensional search space is too small for reasonable comparisons.

The six-dimensional Hartmann function (Dixon and Szego, 1978) is of higher dimension. Like the Branin function, all algorithms perform quite similar in the beginning, only gird search performs significantly worse within the first 35 iterations (see Figure 14b). Yet, in contrast to Branin, no algorithm is able to find the global minimum within 250 iterations, with RoBO yielding the best results. Grid search and random search yield the worst results. The remaining algorithms perform almost the same.

Over all synthetic benchmarks, RoBO was able to consistently outperform or yield equivalent results compared to all competitors. However, absolute differences are small and results vary quite heavily depending on the random state. This implies that synthetic test functions—especially the Branin function which is used quite often in literature to evaluate CASH algorithms, e.g., (Snoek et al., 2012; Eggensperger et al., 2015; Klein et al., 2017)—are not suited as a benchmark as no significant differences can be observed. In addition, the empirical evaluations revealed that the performance of all algorithms heavily relies on the random seed. Therefore, such experiments should always be repeated several times and average performances should be reported.

Besides the performance, also the average runtime of each algorithm per iteration was tracked. The total runtime is dominated by the configuration selection overhead as a single evaluation of a synthetic test function requires less than 0.1 ms. Even though this scenario is very unusual for machine learning, it allows an empirical analysis of the introduced overhead for each algorithm.

Figure 15 shows the computational overhead introduced by each CASH algorithm on the 20 dimensional Rosenbrock function (Dixon and Szego, 1978). Each algorithm was executed without parallelization for 500 iterations. To compensate external influences, the experiment was repeated 20 times and the results are averaged. Grid search, random search and Optunity basically have a constant overhead $\mathcal{O}(1)$ not changing during the 500 iterations.

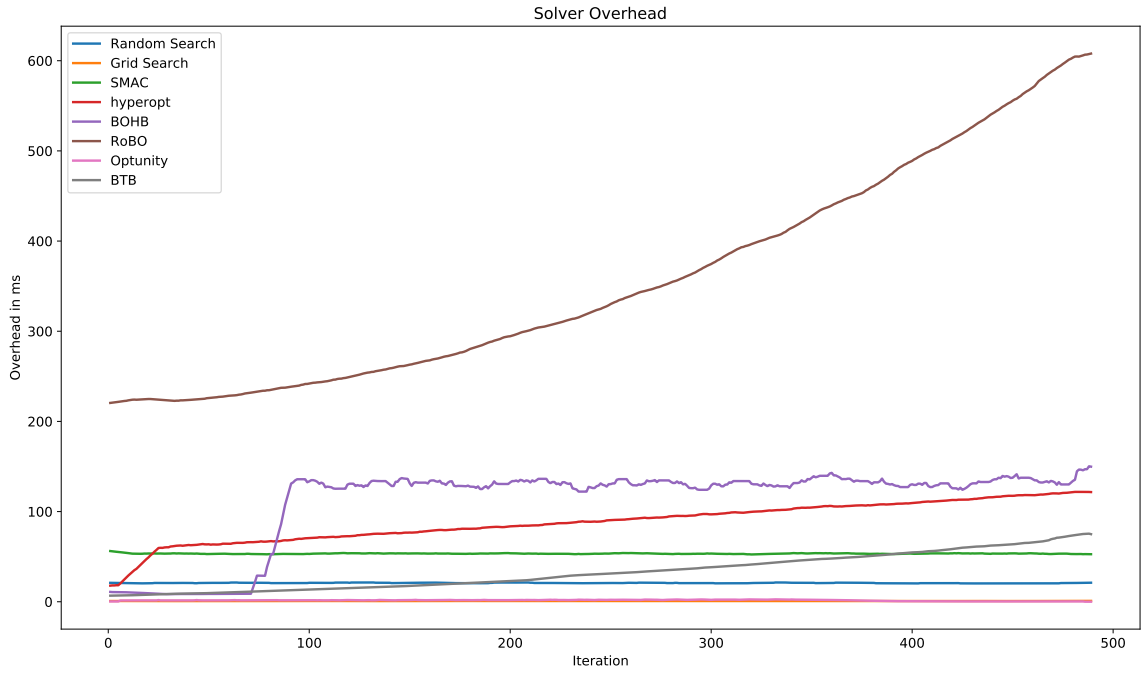


Figure 15: Overhead introduced by different CASH solvers.

This is not surprising as those methods do not store a history of evaluated configurations. Yet, surprisingly SMAC also has a constant overhead. HYPEROPT and BOHB should in theory have a similar overhead of $\mathcal{O}(n)$ as BOHB internally uses an algorithm closely related to HYPEROPT. However, BOHB uses an external component to distribute the optimization in a cluster. Without any parallelization, the synchronization overhead dominates the overhead of HYPEROPT. Finally, RoBO and BTB both use Gaussian processes. The cubic time complexity of Gaussian processes is clearly visible for RoBO and partially for BTB. Due to the usage of Markov chain Monte Carlo (Rubinstein and Kroese, 2008), RoBO has a significantly larger overhead. Except RoBO, all methods have a runtime overhead of less than 200 ms to select a configuration.

With the focus on ML, the introduced overhead of all methods is relatively insignificant. Normally, at most a few hundred different configurations are tested. Consequently, the overhead is only relevant for very cheap objective functions, as it is the case with the considered synthetic test functions, or in combination with extensive usage of low fidelity methods.

Even though synthetic test functions allow very fast evaluation of different configurations as no ML model has to be fitted, they have some severe drawbacks:

1. All synthetic test functions are defined as $f : \mathbb{R}^d \to \mathbb{R}$. Consequently, a simulation of categorical hyperparameters is not possible leading to an unrealistic configuration space. Furthermore, the search space is completely unstructured.

2. The previous experiments have shown that the performance of the different solvers is not clearly distinguishable.

3. Synthetic test functions are limited to HPO as no real data sets are used. Consequently measures to improve the data quality, e.g., data cleaning and feature engineering, are not applicable.

4. All synthetic test functions have a continuous and smooth surface. These properties do not hold for real response surfaces (Eggensperger et al., 2015).

Consequently, even though synthetic test functions are very convenient, they are not well suited for CASH benchmarking.

9.3 Empirical Performance Models

In the previous section it was shown that synthetic test functions are not suited for benchmarking. Using real data sets as an alternative is very inconvenient. Even though they provide the most realistic way to evaluate AutoML algorithms (see Section 9.4 for more details) the time for fitting a single model can become prohibitively large. In order to significantly lower the turnaround time for testing a single configuration, empirical performance models (EPMs) have been introduced (Eggensperger et al., 2015, 2017).

An EPM is a surrogate for a real data set that models the response surface of a specific loss function. By sampling the performance of many different configurations, a regression model of the response surface is created. Using grid search, training samples covering the complete configuration space are generated. Additionally, multiple SMBO algorithms are used to sample performance measures for well performing regions. This step is important as the regression model has to represent the loss function close to local minima very accurately. In general, the training of an EPM is very expensive as several thousand models with different configurations have to be trained.

The benefit of this computational heavy setup phase is that the turnaround time of testing new configurations proposed by an AutoML algorithm is significantly reduced. Instead of training an expensive model, the performance can be retrieved in quasi constant time from the regression model. In combination with the realistic configuration space and response surface, EPMs are very useful.

In theory, EPMs can be used for CASH as well as complete pipeline creation. However, in reality only EPMs for CASH are available. Due to the quasi exhaustive analysis of the configuration space, EPMs heavily suffer from the curse of dimensionality. Even by using a fixed pipeline structure with one preprocessing method and one model and limiting the available algorithms to the ones implemented in auto-sklearn, the resulting configuration space would contain 255 combinations of algorithms with a total number of 108 hyperparameters (Feurer et al., 2015a). Consequently, no EPMs are available to test the performance of a complete ML pipeline.

In the context of this work EPMs have not been evaluated. Instead real data sets have been directly used for performance evaluations.

9.4 Real Data Sets

All previous introduced methods for performance evaluations only focus on the aspect of selecting and tuning a modeling algorithm. Data cleaning and feature engineering are completely ignored even though those two steps have a significant impact on the final

performance of an ML pipeline (Chu et al., 2016). The only possibility to capture and evaluate all aspects of AutoML algorithms is using real data sets. However, real data sets also introduce a significant overhead for evaluation as for each pipeline multiple ML models have to be trained. Depending on the complexity and size of the data set, testing a single pipeline can require several hours of wall clock time. In total multiple months of CPU time were necessary to conduct all evaluations with real data sets presented in this survey.

As explained in Section 2, the performance of an AutoML algorithm depends on the tested data set. Consequently, it is not useful to evaluate performance on only a few data sets in detail but instead the performance is evaluated on a wide range of different data sets. To ensure reproducibility of the results, only publicly available data sets are used. Therefore, data sets from OpenML (Vanschoren et al., 2014), a collaborative platform for sharing data sets in a standardized format, have been selected. More specifically, the curated benchmarking suite OpenML100 (Bischl et al., 2017) is used.[4] This suite contains a list of 100 classification tasks with high quality data sets, meaning: 1. the data sets contain more than 500 and less than $100,000$ records, 2. have less than $5,000$ features and 3. class distributions are not to heavily skewed. However, high-quality does not imply that no preprocessing of the data is necessary as for example some data sets contain missing values.

At first, all previously mentioned CASH algorithms are tested on the complete OpenML100 suite. Therefore, a hierarchical configuration space containing 13 classifiers with total number of 58 hyperparameters is created. This configuration space—listed in Table 4 and Appendix B—is used by all CASH algorithms. Algorithms not supporting hierarchical configuration spaces use a configuration space without conditional dependencies. Furthermore, if no categorical or integer hyperparameters are supported, these parameters are transformed to continuous variables. This transformation is reversed before training a model.

All algorithms are limited to exactly 325 iterations. Preliminary evaluations have show that all algorithm basically always converge before hitting this iteration limit. The model fitting in each iteration is limited to 10 minutes. This timeout cancels roughly 1.4% of all evaluations. Consequently, the influence on the final results is negligible while the overall runtime is reduced by orders of magnitude. Some algorithms, for example RoBO, only support HPO without algorithm selection. For those algorithms, an optimization instance is created for each ML algorithm. The number of iterations per estimator is limited to 25 such that the total number of iterations still equals 325. Next, the optimal configuration for each estimator is calculated independently of the other estimators. Finally, the best performing estimator is selected.

The performance of each configuration is determined by a fixed hold-out validation set (30% of the data set); no cross-validation is implemented. As loss function, the misclassification rate

$$\mathcal{L}_{\text{MR}}(\hat{y}, y) = 1 - \left(\frac{1}{|y|} \sum_{i=1}^{|y|} \mathbb{1}(\hat{y}_i = y_i) \right) \tag{10}$$

is used, with $\mathbb{1}$ being an indicator function. To eliminate the effects of non-determinism, all experiments are repeated ten times and results are averaged. Preliminary tests revealed,

4. Available at `https://www.openml.org/s/14`.

Algorithm	#λ	Cat	Con
Bernoulli naïve Bayes	2	1	1
Multinomial naïve Bayes	2	1	1
Decision Tree	4	1	3
Extra Trees	5	2	3
Gradient Boosting	8	1	5
Random Forest	5	2	4
K Nearest Neighbors	3	2	1
LDA	4	1	3
QDA	1	0	1
Linear SVM	4	2	2
Kernel SVM	7	2	5
Passive Aggressive	4	2	2
Linear Classifier with SGD	10	4	6

Table 4: Configuration space for classification algorithms. In total 13 different algorithms with 58 hyperparameters are available. The number of categorical (Cat), continuous (Con) and total number of hyperparameters (#λ) is listed.

that all algorithms are limited by CPU power and not available memory. No algorithm required more than 4 GB of memory, most of that memory was required by the training of the ML model and not the search algorithm itself. Therefore, the memory consumption is not further considered. All experiments were conducted on Intel Xeon E5 processors with 8 cores and 30 GB memory.

Table 5 contains the results of the OpenML100 evaluation. Reported are the average misclassification rate including standard deviation, the average runtime and percentage of failed configuration evaluations. The OpenML100 suite contains 18 datasets with missing values. As no algorithm in the configuration space (see Table 4) is able to handle missing values, all evaluations on these data sets failed. These data sets are not considered for the evaluations. Furthermore, some configuration evaluations failed due to invalid configurations[5] or due to violating the evaluation timeout of 10 minutes. All failed evaluations are assigned the worst possible result of 1.0.

At first, grid search was evaluated. Each continuous hyperparameter is split into two distinct values leading to $6,206$ different configurations. As the number of evaluations is limited to 325 configurations only the first 10 classifiers are tested completely, Kernel SVM only partially, Passive Aggressive and SGD not at all. As Figure 16 shows, grid search has a very low convergence speed leading to the highest misclassification rate. This behavior is consistent with the synthetic benchmarks. Grid search has the shortest evaluation time as small hyperparameter values are tested first allowing fast convergence of the model fitting.

5. For more details see for example `https://github.com/scikit-learn/scikit-learn/issues/7714`.

Algorithm	Misclassification Rate	Runtime	% Failed
Grid Search	0.1589 ± 0.1269	**0:35:12**	16.91
Random Search	0.1417 ± 0.1209	1:35:57	12.43
RoBO	0.0919 ± 0.1223	3:30:20	03.22
hyperopt	**0.0893 ± 0.1203**	5:28:55	07.81
SMAC	0.0903 ± 0.1197	2:27:59	07.43
BOHB	0.0921 ± 0.1204	1:03:45	06.19
BTB	0.0928 ± 0.1215	2:42:38	**01.97**
Optunity	0.1320 ± 0.1228	4:04:07	20.77

Table 5: Results of all tested CASH solvers on the OpenML100 benchmark suite. This table shows the average misclassification rate, runtime in hours and the percentage of failed evaluations per CASH algorithm.

Random search performs better than grid search but still significantly worse than all guided search strategies. Invalid configurations are tested multiple times as no knowledge of bad performing regions is exploited leading to the high percentage of failed evaluations. As random search does virtually not contain any overhead, it is save to assume that on average a single data set requires 1.5 hours for evaluation. Based on this runtime, the overhead or tendency to select long running configurations of the guided approaches can be evaluated.

As expected, all guided algorithms were able to outperform the classic HPO strategies grid and random search. These approaches, with the exception of Optunity, have a very similar misclassification rate—on average 0.0912 ± 0.1208—with a maximum difference of 0.0035. For those algorithms, the runtime and percentage of failed evaluations is more important to distinguish the overall performance. hyperopt yields the overall lowest misclassification rate with 0.0893. However, it also has the longest runtime with roughly 5.5 hours per data set. This is quite surprising as Figure 15 would suggest that RoBO has the longest runtime. A possible explanation is that hyperopt tends to select configurations requiring a long training time. RoBO has a surprisingly competitive misclassification rate of 0.0919. The inability of Gaussian processes to handle categorical variables would have suggested a subpar performance. Additionally, only 3.22% of all evaluations failed. Yet, RoBO introduces also a quite high runtime of 3.5 hours. This effect is limited by BTB. By using multiple Gaussian processes, the average runtime can be decreased by 22.86% in comparison to RoBO. The misclassification rate is marginally worse while the number of failed evaluations is only 1.97%. SMAC has an average performance regarding all criteria. BOHB is the only algorithm making heavy usage of multi-fidelity approximations. These approximations are reflected in an average runtime of 1 hour, being even faster than random search. Yet, the misclassification rate is still similar to the other approaches. Finally, Optunity performs worst of all guided algorithms. Optunity cannot handle categorical hyperparameters very well as the particle movements often violate the constraints of the categorical hyperparameters. The randomly generated first generation is evaluated completely, and the following generations are often aborted. Consequently, less valid configurations are evaluated leading to the subpar misclassification rate of 0.1320. Furthermore, Optunity has the second highest runtime with over 4 hours.

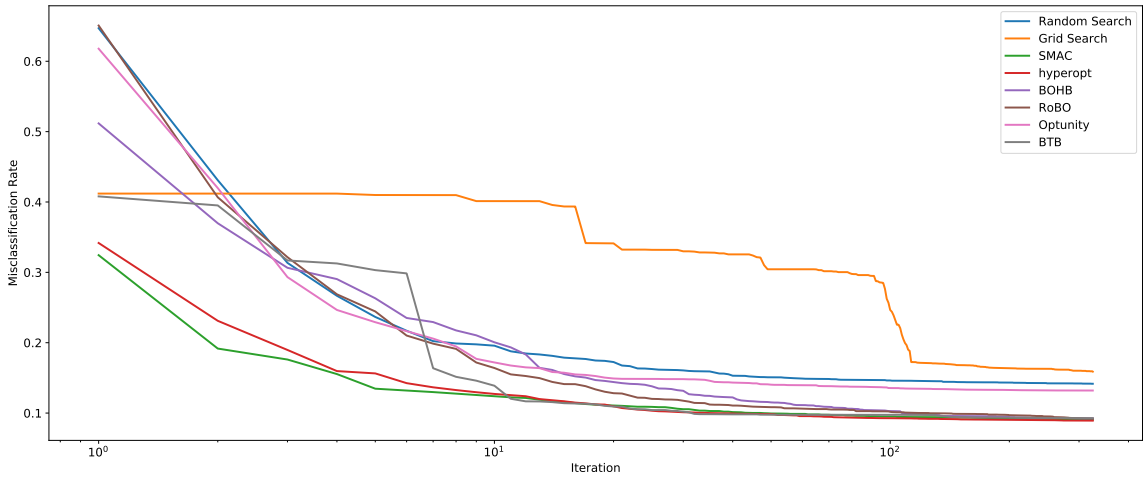


Figure 16: Average misclassification rate of all CASH solvers on the OpenML100 suite.

Figure 16 shows the average incumbent performance for all tested algorithms over 325 iterations. In general, the performance of all algorithms converges around 200 iterations. Afterwards the misclassification rate is on average only minimally reduced from 0.1134 to 0.1111. For the first ten iterations, SMAC and hyperopt significantly outperform all other algorithms. After roughly ten iterations BTB has a similar average misclassification rate. BOHB and RoBO yield a similar performance after roughly 100 iterations.

The OpenML100 suite only contains medium sized data sets. However, the results from this evaluation can also be transferred to large data sets. Only the timeout for a single evaluation has to be adjusted to incorporate the longer fitting duration.

BOHB provides the best compromise for algorithm selection and HPO. Due to the multi-fidelity approximations, BOHB is able to test many configurations in a short time while yielding good results on both synthetic test functions and real data sets. If the final performance is very important and training duration is neglectable, RoBO consistently provides outstanding results on the synthetic benchmarks and good results for real data sets.

Finally, AutoML frameworks capable of building complete ML pipelines are evaluated. Therefore, 20 different data sets from the OpenML100 suite have been selected: at first, all tasks from the previous CASH evaluation with an average misclassification rate of over 30% have been selected to evaluate the benefits of using a complete pipeline instead of only a classifier. Furthermore, 15 data sets from OpenML100 containing missing values have been selected. All selected data sets are listed in Table 6. To limit the effects of non-determinism, all experiments are repeated 10 times and results are averaged. Each framework is configured to use its default configuration space. Similar to the previous experiments, the pipeline performance is determined by a fixed hold-out validation set (30% of the data set) and the misclassification rate in Equation (10) is used as the loss function. Each evaluation was limited to four hours.

Tested are ATM, auto-sklearn, hyperopt-sklearn and TPOT. Additionally, the auto-sklearn pipeline structure with a random preprocessor and classifier is included as random search. auto_ml does not provide any measures to stop the tuning after a

fixed number of iterations or a time budget. Therefore, auto_ml is not considered for the empirical evaluation. Similarly, ML-Plan does not support scikit-learn as an ML library. To prevent skewed results, ML-Plan is also not evaluated. hyperopt-sklearn does not support a time budget. Using preliminary tests, the number of iterations is limited to 325 leading to maximum runtimes of just under 4 hours.

As Table 6 indicates, auto-sklearn and TPOT yield similar results for nearly all data sets with TPOT being marginally better. As a consequence, TPOT has the best results on 50% of all tested data sets and also the best overall performance with a misclassification rate of 0.2294. auto-sklearn yields the best result on 25% of all data sets and the second best average performance. ATM occasionally suffers from the missing evaluation timeout. Fitting of an unfavorable configuration sometimes requires several hours. Consequently, fewer than possible configurations are evaluated and ATM violates the overall time budget. However, no new configurations are tested in parallel as soon as the budget is exhausted. Therefore, ATM does not have an unfair advantage. The evaluation of data sets 1112, 1590 and 23512 consistently exceeded the available memory while evaluations of data set 23380 always fail due to numeric overflows. Data sets 470 and 4538 failed on 50% of all evaluations without an apparent reason. Furthermore, the performance on data set 1475 and 1492 is so poor that the learning basically failed. On the remaining data sets, ATM's performance is similar to or even surpasses auto-sklearn and TPOT.

hyperopt-sklearn has the most problems evaluating the data sets: 1. hyperopt-sklearn uses a very basic pipeline structure with exactly one preprocessor and classifier. Measures for data imputation are not included. Consequently, hyperopt-sklearn fails to evaluate all data sets containing missing values. 2. No parallel evaluation of configurations is implemented. Therefore, only ⅛ of the possible configurations are evaluated. 3. For no apparent reason, multiple evaluations of various data sets failed. Those cases are treated as a misclassification rate of 1.0. As a consequence, hyperopt-sklearn is able to process only five data sets. Of those five data sets, only two have a similar performance to the other frameworks. Finally, random search is evaluated as a baseline comparison. Random search is always—with the exception of one data set with a marginal difference in the final performance—outperformed by auto-sklearn and TPOT. ATM is outperformed by random search on eight data sets (including the data sets where ATM was not able to find a solution). Finally, random search is able to yield better results than hyperopt-sklearn on all except a single data set. As ATM and hyperopt-sklearn fail on many tasks, the average performance is very low. In contrast, random search is able to always yield a working pipeline. In consequence, random search outperforms those two frameworks on average.

Even though TPOT had the best overall performance, auto-sklearn is currently the best compromise for building complete ML pipelines automatically. The average performance is similar to TPOT, yet auto-sklearn has a higher convergence speed than TPOT allowing building good pipelines in a fraction of training time.

Finally, Table 7 compares the average performance of single classifiers selected via CASH with a complete pipeline on the five datasets with an original misclassification rate of over 30%. Results of the complete pipeline are filtered to remove obviously unsuccessful optimizations. It is apparent that the average pipeline performance is always worse than the single classifiers. This result coincides with Schoenfeld et al. (2018) stating that after adding

OpenML Data Set		auto-sklearn	TPOT	ATM	hyperopt-sklearn	Random Search
breast-w	(15)	02.62 ± 01.79	02.98 ± 02.29	**01.44 ± 02.22**	–	09.04 ± 06.66
cmc *	(23)	45.14 ± 30.78	**43.33 ± 01.52**	45.14 ± 04.53	46.53 ± 01.41	54.34 ± 13.03
mushroom	(24)	**00.00 ± 00.00**	**00.00 ± 00.00**	**00.00 ± 00.00**	–	**00.00 ± 00.00**
credit-approval	(29)	13.41 ± 05.15	13.41 ± 02.61	**11.11 ± 03.52**	–	15.34 ± 04.47
sick	(38)	01.74 ± 01.17	**01.30 ± 00.87**	02.72 ± 00.87	–	11.72 ± 10.13
soybean	(42)	07.79 ± 00.88	07.07 ± 02.79	**06.59 ± 02.79**	–	08.09 ± 05.65
eucalyptus	(188)	**35.41 ± 05.02**	35.75 ± 04.00	38.17 ± 05.24	–	45.74 ± 10.45
irish	(451)	**00.00 ± 00.00**	**00.00 ± 00.00**	**00.00 ± 00.00**	–	**00.00 ± 00.00**
analcatdata_dmf *	(469)	77.71 ± 08.54	79.48 ± 01.86	**75.10 ± 02.59**	80.83 ± 04.50	79.49 ± 01.33
profb	(470)	34.78 ± 06.57	**32.43 ± 05.84**	64.11 ± 89.83	–	45.14 ± 13.46
jm1	(1053)	18.84 ± 01.26	**18.13 ± 01.55**	24.56 ± 16.05	–	18.59 ± 02.06
KDDCup09_churn	(1112)	07.30 ± 00.22	**07.24 ± 00.29**	–	–	18.43 ± 10.06
first-order-theo. *	(1475)	40.29 ± 02.06	**38.48 ± 01.95**	60.86 ± 39.37	55.33 ± 12.67	48.70 ± 06.22
plants-shape *	(1492)	**36.54 ± 03.53**	41.25 ± 06.61	92.03 ± 14.92	63.33 ± 13.02	45.93 ± 10.52
adult	(1590)	13.10 ± 00.83	**13.03 ± 00.29**	–	–	13.33 ± 03.87
GestureSeg. *	(4538)	33.13 ± 02.16	**32.78 ± 03.59**	85.31 ± 36.73	46.50 ± 12.29	43.77 ± 10.02
cylinder-bands	(6332)	27.01 ± 06.40	19.91 ± 04.55	**17.75 ± 03.99**	–	22.84 ± 08.45
cjs	(23380)	03.16 ± 03.03	**00.00 ± 00.00**	–	–	00.36 ± 00.76
dresses-sales	(23381)	44.67 ± 14.19	44.00 ± 08.74	**32.67 ± 05.89**	–	46.83 ± 09.53
higgs	(23512)	**27.54 ± 00.73**	28.25 ± 00.75	–	–	28.40 ± 06.77
Average		23.51 ± 03.37	**22.94 ± 02.51**	34.85 ± 14.29	58.51 ± 15.98	27.80 ± 05.74

Table 6: Misclassification rate in percent of AutoML frameworks on selected OpenML data sets. Data sets marked by * had an average misclassification rate greater than 30%. Entries marked by – consistently failed to generate an ML pipeline.

OpenML Data Set		CASH	Pipeline
cmc	(23)	**0.4055**	0.4889
analcatdata_dmft	(469)	**0.7441**	0.7852
first-order-theorem	(1475)	**0.3848**	0.4570
plants-shape	(1492)	**0.3988**	0.4676
GestureSegmentation	(4538)	**0.3163**	0.3904

Table 7: Average performance of single classifiers selected via CASH and complete ML pipelines on selected OpenML data sets.

preprocessing steps on average the performance decreases. However, the small decrease in performance (on average 0.0679) allows processing data sets with lower quality.

In summary, the previous experiments with complete ML pipelines revealed that the pipeline structure is crucial for the overall performance. Using only a single classifier, hyperopt was able to obtain the best results, however, hyperopt-sklearn has the worst results due to the very rudimentary pipeline structure used. Consequently, fixed pipeline structures should be selected very carefully. An example for a good pipeline structure is auto-sklearn. Even in combination with random search this fixed pipeline structure is very robust while still yielding moderate results. Furthermore, the previous experiments, including CASH as well as complete pipelines, have shown that current AutoML frameworks have very similar average performances. This often contradicts the performances stated in the original works. Consequently, it is important to have standardized data sets for performance measures to tackle misleading or amended performance evaluations and to get fair comparisons.

10. Discussion and Opportunities for Future Research

Currently, AutoML is completely focused on supervised learning. Even though some methods may be applicable for unsupervised or reinforcement learning, researchers always test their proposed approaches for supervised learning. Furthermore, all existing high-level AutoML frameworks only support classification and regression tasks. Dedicated research for unsupervised or reinforcement learning could boost the development of AutoML framework for currently uncovered learning problems. Additionally, specialized methods could improve the performance for those tasks.

The majority of all publications currently treats the CASH problem either by introducing new solvers or adding performance improvements to existing approaches. A possible explanation could be that CASH is completely domain-agnostic and therefore comparatively easier to automate. However, CASH is only a small piece of the puzzle to build an ML pipeline automatically. A data scientist usually spends 60–80% of his time with cleaning a data set and feature engineering (Pyle, 1999) and only 4% with fine tuning of algorithms (Press, 2016). This distribution is currently not reflected in research efforts. We have not been able to find any literature covering advanced data cleaning methods. Regarding feature creation, all methods are based upon deep feature synthesis. For building flexible pipelines currently only three different approaches have been proposed. Further research

in any of these three areas can highly improve the overall performance of an automatically created ML pipeline.

So far, researchers have focused on a single point of the pipeline creation process. Combining dynamically shaped pipelines with automatic feature engineering and sophisticated CASH methods has the potential to beat the currently available frameworks. However, the complexity of the search space is raised to a whole new level probably requiring new methods for efficient search. Nevertheless, the long term goal should be automatically building complete pipelines with every single component optimized.

AutoML aims to completely automate the creation of an ML pipeline to enable domain expert to use ML. Except very few publications, e.g., (Friedman and Markovitch, 2015; Smith et al., 2017), current AutoML algorithms are designed as a black-box. Even though this may be convenient for an inexperienced user, this approach has two major drawbacks:

1. A domain expert has a profound knowledge about the data set. Using this knowledge, the search space can be significantly reduced.

2. Interpretability of ML has become more important in recent years (Doshi-Velez and Kim, 2017). Users want to be able to understand how a model has been obtained. Using hand-crafted ML models, the reasoning of the model is often already unknown to the user. By automating the creation, the user basically has no chance to understand why a specific pipeline has been selected.

Human-guided ML (Langevin et al., 2018; Gil et al., 2019) aims to present simple questions to the domain expert to guide the exploration of the search space. The domain expert would be able to guide model creation by his experience. Further research in this area may lead to more profound models depicting the real-world dependencies closer. Simultaneously, the domain expert could have the chance to better understand the reasoning of the ML model. This could increase the acceptance of the proposed pipeline.

AutoML frameworks usually introduce their own hyperparameters that can be tuned by an user. Yet, this is basically the same problem that AutoML tried to solve in the first place. Research leading to frameworks with less hyperparameters is desirable (Feurer and Hutter, 2018).

11. Conclusion

In this paper, we have introduced the notation of the pipeline creation problem as a minimization problem. Furthermore, we presented and evaluated various methods for automating each step of creating an ML pipeline. Finally, extensive evaluations of the most popular AutoML tools have been performed on publicly available data sets.

The topic AutoML has come a long way since its beginnings in the 1990s. Especially in the last eight years, it has received a lot of attention from research, enterprises and media. Current state-of-the-art frameworks enable domain experts building reasonable well performing ML pipelines without knowledge about ML or statistics. Seasoned data scientists can profit from the automation of tedious manual tasks, especially model selection and HPO. However, automatically generated pipelines are not able to beat human experts yet (Guyon et al., 2016). It is likely, that AutoML will continue to be a hot research topic leading to even better, holistic AutoML frameworks in the near future.

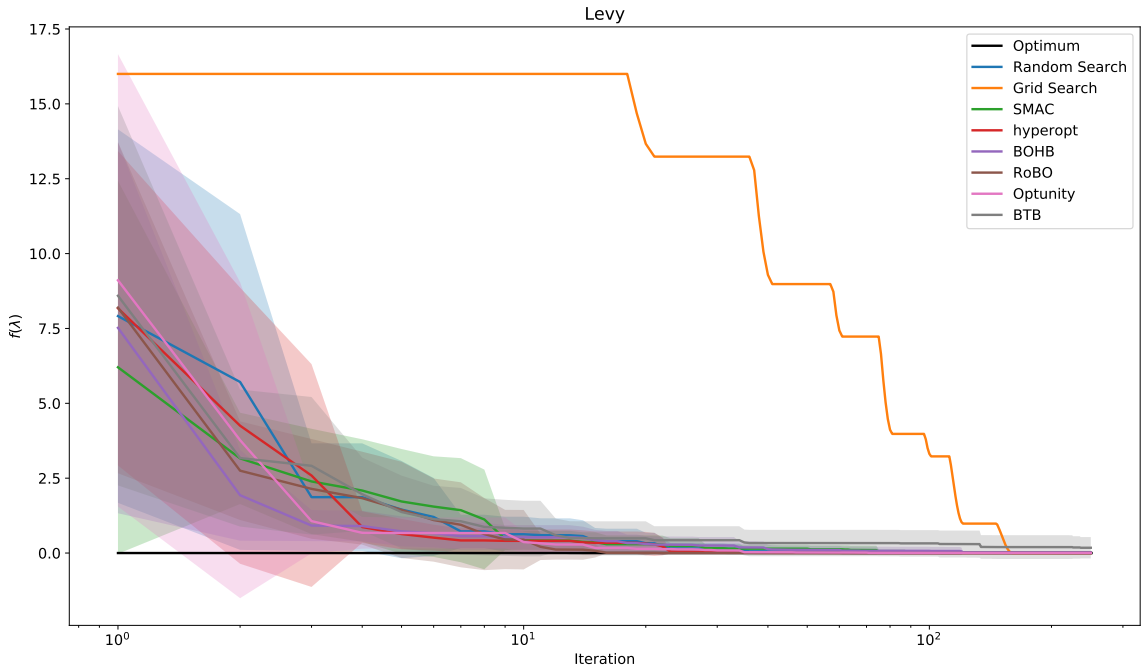


Figure 17: Minimization progress of various CASH algorithms on the Levy function.

Acknowledgments

This work is partially supported by the Federal Ministry of Transport and Digital Infrastructure within the mFUND research initiative and the Ministry of Economic Affairs of the state Baden-Württemberg within the Center for Cyber Cognitive Intelligence.

Appendix A. Synthetic Benchmark Results

Figure 17 shows the average results for all CASH algorithms on the one-dimensional Levy function (Laguna and Martí, 2005). All algorithms except BOHB and BTB are able to find the exact global minimum. BOHB and BTB miss the global optimum with a small offset (see Table 3). The advanced algorithms and random search converge significantly faster to the global minimum than grid search. Regarding the advanced algorithms a ranking of the performance is not possible as no significant differences in the mean incumbent are visible. Furthermore, these algorithms consistently converge within roughly 40 iterations leading to a small standard deviation. Only BOHB has a larger variance due to the fixed percentage of random evaluations.

Figure 18 shows the average results for all CASH algorithms on the two-dimensional Camelback function (Molga and Smutnicki, 2005). The first 17 iterations of grid search, with a score between 2400 and 6400, are not displayed for better readability. All algorithms are able to converge to the global minimum after 30 iterations; only grid search requires 110 iterations to converge. Furthermore, grid search performs significantly worse than all other algorithms for the complete optimization period. The results of the remaining

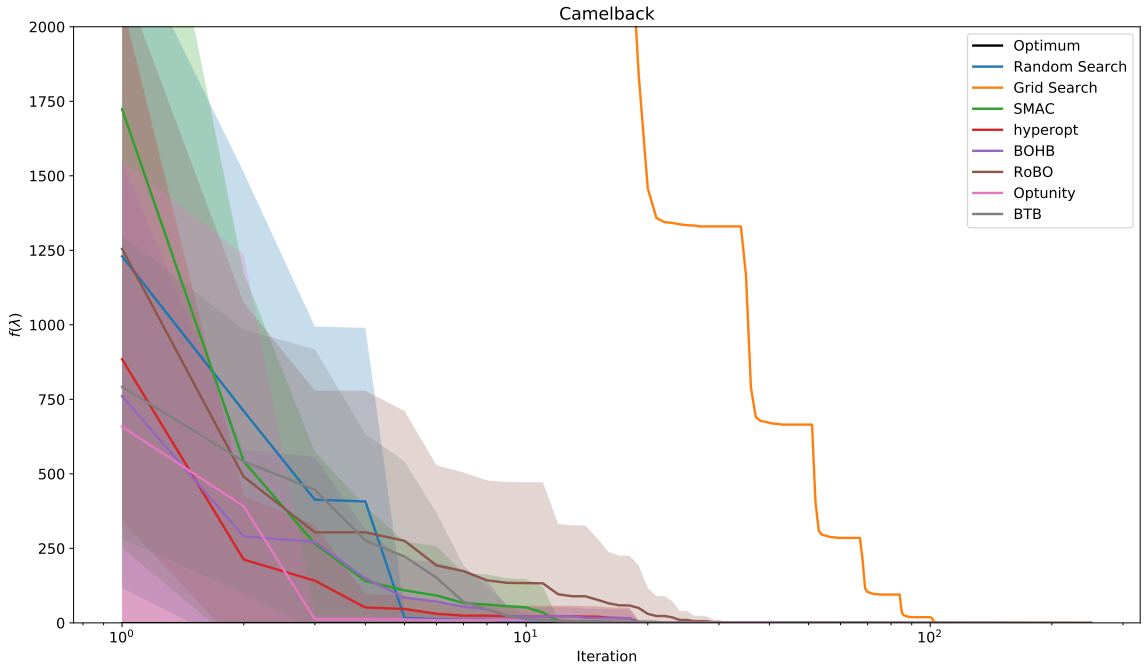


Figure 18: Minimization progress of various CASH algorithms on the Camelback function.

algorithms are all very similar and a reliable statement which algorithm performs best is difficult. Random search, SMAC and BOHB yield good scores of 30. RoBO, hyperopt, BTB and Optunity yield scores of roughly 5 getting quite close to the global minimum of -1.0316.

Figure 19 shows the average results for all CASH algorithms on the ten-dimensional Rosenbrock function (Dixon and Szego, 1978). Grid search only evaluates two values per dimension, yet still randomly selects the very first configuration close to the global optimum. Consequently, grid search outperforms all other algorithms for many iterations with only RoBO being able to find a better configuration after 80 iterations. Regarding all other algorithms, no algorithm gets close to the global optimum. All guided procedures have a very high standard deviation and therefore no final answer can be provided which algorithm performs better. Random search yields the worst average performance. Optunity, SMAC and BOHB are on average on par; hyperopt and BTB perform better but still worse than grid search. In general, no algorithm is able to converge close to the global optimum within 250 iterations, suggesting that 250 iterations are not sufficient for such a high-dimensional search space.

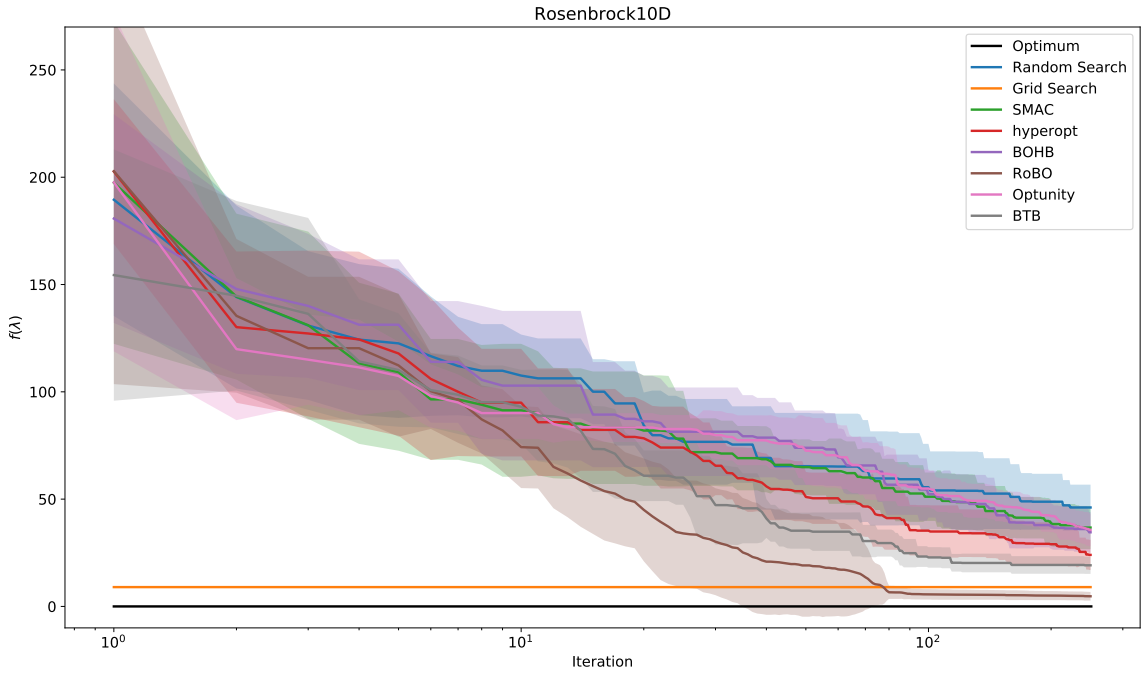


Figure 19: Minimization progress of various CASH algorithms on the Rosenbrock function.

Appendix B. Configuration Space for CASH Solvers

Classifier	Hyperparameter	Type	Values
Bernoulli naïve Bayes			
	alpha	con	[0.01, 100]
	fit_prior	cat	[false, true]
Multinomial naïve Bayes			
	alpha	con	[0.01, 100]
	fit_prior	cat	[false, true]
Decision Tree			
	criterion	cat	[entropy, gini]
	max_depth	int	[1, 10]
	min_samples_leaf	int	[1, 20]
	min_samples_split	int	[2, 20]
Extra Trees			
	bootstrap	cat	[false, true]
	criterion	cat	[entropy, gini]
	max_features	con	[0.0, 1.0]
	min_samples_leaf	int	[1, 20]
	min_samples_split	int	[2, 20]

Gradient Boosting			
	learning_rate	con	[0.01, 1.0]
	criterion	cat	[friedman_mse, mae, mse]
	max_depth	int	[1, 10]
	min_samples_split	int	[2, 20]
	min_samples_leaf	int	[1, 20]
	n_estimators	int	[50, 500]
Random Forest			
	bootstrap	cat	[false, true]
	criterion	cat	[entropy, gini]
	max_features	con	[0.0, 1.0]
	min_samples_split	int	[2, 20]
	min_samples_leaf	int	[1, 20]
	n_estimators	int	[2, 100]
k Nearest Neighbors			
	n_neighbors	int	[1, 100]
	p	int	[1, 2]
	weights	cat	[distance, uniform]
LDA			
	n_components	cat	[1, 250]
	shrinkage	con	[0.0, 1.0]
	solver	cat	[eigen, lsgr, svd]
	tol	con	[0.00001, 0.1]
QDA			
	reg_param	con	[0.0, 1.0]
Linear SVM			
	C	con	[0.01, 10000]
	loss	cat	[hinge, squared_hinge]
	penalty	cat	[l1, l2]
	tol	con	[0.00001, 0.1]
Kernel SVM			
	C	con	[0.01, 10000]
	coef0	con	[-1, 1]
	degree	int	[2, 5]
	gamma	con	[1, 10000]
	kernel	cat	[poly, rbf, sigmoid]
	shrinking	cat	[false, true]
	tol	con	[0.00001, 0.1]
Passive Aggressive			
	average	cat	[false, true]
	C	con	[0.00001, 10]
	loss	cat	[hinge, squared_hinge]
	tol	con	[0.00001, 0.1]

SGD			
	alpha	con	[0.0000001, 0.1]
	average	cat	[false, true]
	epsilon	con	[0.00001, 0.1]
	eta0	con	[0.0000001, 0.11]
	learning_rate	cat	[constant, invscaling, optimal]
	loss	cat	[hinge, log, modified_huber]
	l1_ratio	con	[0.0000001, 1]
	penalty	cat	[elasticnet, l1, l2]
	power_t	con	[0.00001, 1]
	tol	con	[0.00001, 0.1]

Table 8: Complete configuration space used for OpenML100 CASH solvers. Hyperparameter names equal the used names in scikit-learn. *cat* are categorical, *con* are continuous and *int* integer hyperparameters.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

Rodrigo Agerri, Josu Bermudez, and German Rigau. IXA pipeline: Efficient and Ready to Use Multilingual NLP tools. In *Language Resources and Evaluation*, pages 3823–3828, 2014.

Shawkat Alia and Kate A. Smith-Miles. A meta-learning approach to automatic kernel selection for support vector machines. *Neurocomputing*, 70(1-3):173–186, 2006.

Richard Loree Anderson. Recent Advances in Finding Best Operating Conditions. *Journal of the American Statistical Association*, 48(264):789–798, 1953.

Peter Auer. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2002.

Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a Rigged Casino: The Adversarial Multi-Armed Bandit Problem. *Foundations of Computer Science*, pages 322–346, 1995.

Pourya Ayria. A complete Machine Learning PipeLine, 2018. URL `https://www.kaggle.com/pouryaayria/a-complete-ml-pipeline-tutorial-acu-86`.

Baidu. EZDL, 2018. URL `http://ai.baidu.com/ezdl/`.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. *CoRR*, abs/1611.0:1–18, 2016.

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1997.

Yoshua Bengio. Gradient-Based Optimization of Hyperparameters. *Neural Computation*, 12(8):1889–1900, 2000.

James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *International Conference on Neural Information Processing Systems*, pages 2546–2554, 2011.

James Bergstra, Dan Yamins, and David D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Python in Science Conference*, pages 13–20, 2013.

Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. OpenML Benchmarking Suites and the OpenML100. *arXiv preprint arXiv:1708.03731*, 2017.

Leon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *International Conference on Computational Statistics*, pages 177–186, 2010.

Leon Bottou. Stochastic Gradient Descent Tricks. In *Neural Networks, Tricks of the Trade, Reloaded*, pages 430–445. Springer, 2012.

Felix Brandt, Felix Fischer, and Paul Harrenstein. On the Rate of Convergence of Fictitious Play. In *International Symposium on Algorithmic Game Theory*, pages 102–113, 2010.

Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olsen. *Classification and Regression Trees*. Chapman and Hall, 1984.

Eric Brochu, Vlad M. Cora, and Nando de Freitas. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *CoRR*, abs/1012.2, 2010.

Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Senior Member, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.

Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, 1999.

Tobe Chan. Advisor, 2017. URL `https://github.com/tobegit3hub/advisor`.

Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. Autostacker: A Compositional Evolutionary Learning System. *CoRR*, abs/1803.0, 2018.

Peng-Wei Chen, Jung-Ying Wang, and Hahn-Ming Lee. Model selection of SVMs using GA approach. In *IEEE International Joint Conference on Neural Networks*, 2004.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System Tianqi. In *ACM International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *ACM International Conference on Management of Data*, pages 1247–1261, 2015.

Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. Data Cleaning: Overview and Emerging Challenges. In *International Conference on Management of Data*, pages 2201–2206, 2016.

Marc Claesen, Jaak Simm, Dusan Popovic, Yves Moreau, and Bart De Moor. Easy Hyperparameter Search Using Optunity. *CoRR*, abs/1412.1, 2014.

Alibaba Clouder. Shortening Machine Learning Development Cycle with AutoML, 2018. URL `https://www.alibabacloud.com/blog/shortening-machine-learning-development-cycle-with-automl_594232`.

Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*, volume 5. Springer, 2007.

Jason Cooper and Chris Hinde. Improving genetic algorithms' efficiency using intelligent fitness functions. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 636–643, 2003.

Silvia Cristina Nunes das Dôres, Carlos Soares, and Duncan Ruiz. Bandit-Based Automated Machine Learning. In *Brazilian Conference on Intelligent Systems*, 2018.

Manoranjan Dash and Huan Liu. Feature Selection for Classification. *Intelligent Data Analysis*, 1:131–156, 1997.

Péricles B.C. De Miranda, Ricardo B.C. Prudêncio, Andre Carlos P.L.F. De Carvalho, and Carlos Soares. An Experimental Study of the Combination of Meta-Learning with Particle Swarm Algorithms for SVM Parameter Selection. *International Conference on Computational Science and Its Applications*, pages 562–575, 2012.

Alex G. C. de Sá, Walter José G. S. Pinto, Luiz Otávio V. B. Oliveira, and Gisele L. Pappa. RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines. In *European Conference on Genetic Programming*, volume 10196, pages 246–261, 2017.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Thomas Desautels, Andreas Krause, and Joel W. Burdick. Parallelizing Exploration-Exploitation Tradeoffs with Gaussian Process Bandit Optimization. *Journal of Machine Learning Research*, 15:4053–4103, 2014.

Thomas Dinsmore. Automated Machine Learning: A Short History, 2016. URL `https://blog.datarobot.com/automated-machine-learning-short-history`.

L. C. W. Dixon and G. P. Szego. The Global Optimization Problem: An Introduction. *Towards Global Optimisation*, pages 1–15, 1978.

Finale Doshi-Velez and Been Kim. Towards A Rigorous Science of Interpretable Machine Learning. *arXiv preprint arXiv:1702.08608*, 2017.

Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni de Paula Lourenco, Jorge Piazentin Ono, Kyunghyun Cho, Claudio Silva, and Juliana Freire. AlphaD3M : Machine Learning Pipeline Synthesis. In *International Conference on Machine Learning AutoML Workshop*, 2018.

Valeria Efimova, Andrey Filchenkov, and Viacheslav Shalamov. Fast Automated Selection of Learning Algorithm And its Hyperparameters by Reinforcement Learning. In *International Conference on Machine Learning AutoML Workshop*, 2017.

Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters. In *NIPS Workshop on Bayesian Optimization in Theory and Practice*, 2013.

Katharina Eggensperger, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Efficient Benchmarking of Hyperparameter Optimizers via Surrogates. In *AAAI Conference on Artificial Intelligence*, pages 1114–1120, 2015.

Katharina Eggensperger, Marius Thomas Lindauer, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Efficient Benchmarking of Algorithm Configuration Procedures via Model-Based Surrogates. *CoRR*, abs/1703.1, 2017.

Hugo Jair Escalante, Manuel Montes, and Villaseñor Luis. Particle Swarm Model Selection for Authorship Verificatio. *Iberoamerican Congress on Pattern Recognition*, pages 563–570, 2009.

Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *International Conference on Machine Learning*, pages 1437–1446, 2018.

Matthias Feurer and Frank Hutter. Towards Further Automation in AutoML. In *International Conference on Machine Learning AutoML Workshop*, 2018.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenber, Manuel Blum, and Frank Hutter. Efficient and Robust Automated Machine Learning. In *International Conference on Neural Information Processing Systems*, pages 2755–2763, 2015a.

Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. *National Conference on Artificial Intelligence*, pages 1128–1135, 2015b.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Practical Automated Machine Learning for the AutoML Challenge 2018. *International Conference on Machine Learning AutoML Workshop*, 2018.

Jerome H Friedman. On Bias, Variance, 0/1-Loss, and the Curse-of-Dimensionality. *Data Mining and Knowledge Discovery*, 1:55–77, 1997.

Lior Friedman and Shaul Markovitch. Recursive Feature Generation for Knowledge-based Learning. *Journal of Artificial Intelligence Research*, 1:3–17, 2015.

Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes. In *International Conference on Machine Learning AutoML Workshop*, 2017.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition*, 2012.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planmning: Theory & Praxis*. Morgan Kaufmann Publishers, Inc., 2004.

Yolanda Gil, Ke-Thia Yao, Varun Ratnakar, Daniel Garijo, Greg Ver Steeg, Pedro Szekely, Rob Brekelmans, Mayank Kejriwal, Fanghao Luo, and I-Hui Huang. P4ML: A Phased Performance-Based Pipeline Planner for Automated Machine Learning. In *International Conference on Machine Learning AutoML Workshop*, pages 1–8, 2018.

Yolanda Gil, James Honaker, Shikhar Gupta, Yibo Ma, Vito D Orazio, Daniel Garijo, Shruti Gadewar, Qifan Yang, and Neda Jahanshad. Towards Human-Guided Machine Learning. In *International Conference on Intelligent User Interfaces*, 2019.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *ACM International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495, 2017.

Google. Google Trends, 2019. URL `https://trends.google.de/trends/explore?date=today5-y&q=automl`.

Laura Gustafson. *Bayesian Tuning and Bandits : An Extensible , Open Source Library for AutoML by*. PhD thesis, Massachusetts Institute of Technology, 2018.

Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

Isabelle Guyon, Amir Saffari, Gideon Dror, and Gavin Cawley. Analysis of the IJCNN 2007 Agnostic Learning vs. Prior Knowledge Challenge. *Neural Networks*, 21(2-3):544–550, 2008.

Isabelle Guyon, Kristin Bennett, Gavin Cawley, Hugo Jair Escalante, Sergio Escalera, Tin Kam Ho, Núria Maciá, Bisakha Ray, Mehreen Saeed, Alexander Statnikov, and Evelyne Viegas. Design of the 2015 ChaLearn AutoML Challenge. *International Joint Conference on Neural Networks*, pages 1–8, 2015.

Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Maciá, Bisakha Ray, Lukasz Romaszko, Michéle Sebag, Alexander Statnikov, Sébastien Treguer, and Evelyne Viegas. A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention. In *International Conference on Machine Learning AutoML Workshop*, pages 21–30, 2016.

Isabelle Guyon, Lisheng Sun-Hosoya, Marc Boullé, Hugo Jair Escalante, Sergio Escalera, Zhengying Liu, Damir Jajetic, Bisakha Ray, Mehreen Saeed, Michele Sebag, Alexander Statnikov, Wei-Wei Tu, and Evelyne Viegas. Analysis of the AutoML Challenge series 2015-2018. In *Automatic Machine Learning: Methods, Systems, Challenges*. Springer Verlag, 2018.

H2O.ai. H2O Driverless AI, 2018. URL `https://www.h2o.ai/products/h2o-driverless-ai/`.

Joseph M. Hellerstein. Quantitative Data Cleaning for Large Databases. *United Nations Economic Commission for Europe*, 2008.

Philipp Hennig and Christian J. Schuler. Entropy Search for Information-Efficient Global Optimization. *Journal of Machine Learning Research*, 13:1809–1837, 2012.

Jacob Y. Hesterman, Luca Caucci, Matthew A. Kupinski, Harrison H. Barrett, and Lars R. Furenlid. Maximum-Likelihood Estimation With a Contracting-Grid Search Algorithm. *IEEE Transactions on Nuclear Science*, 57(3):1077–1084, 2010.

Matthew W Hoffman, Bobak Shahriari, and Nando de Freitas. On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning. In *Artificial Intelligence and Statistics*, pages 365–374, 2014.

Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification, 2003.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

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

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. An Efficient Approach for Assessing Hyperparameter Importance. In *International Conference on Machine Learning*, pages 754–762, 2014.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Hyperparameter Optimization. In *Automatic Machine Learning: Methods, Systems, Challenges*, pages 3–38. Springer, 2018.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. *CoRR*, abs/1502.0, 2015.

Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative Support for Sensor Data Cleaning. In *International Conference on Pervasive Computing*, pages 83–100, 2006.

Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabás Póczos. Asynchronous Parallel Bayesian Optimisation via Thompson Sampling. In *International Conference on Machine Learning AutoML Workshop*, 2017.

James Max Kanter and Kalyan Veeramachaneni. Deep Feature Synthesis: Towards Automating Data Science Endeavors. In *IEEE International Conference on Data Science and Advanced Analytics*, pages 1–10, 2015.

Gilad Katz, Eui Chul Richard Shin, and Dawn Song. ExploreKit: Automatic feature generation and selection. In *IEEE International Conference on Data Mining*, pages 979–984, 2017.

Ambika Kaul, Saket Maheshwary, and Vikram Pudi. AutoLearn - Automated Feature Generation and Selection. In *IEEE International Conference on Data Mining*, 2017.

Balázs Kégl. How to Build a Data Science Pipeline, 2017. URL `https://www.kdnuggets.com/2017/07/build-data-science-pipeline.html`.

James Kennedy and Russell Eberhart. Particle Swarm Optimization. In *International Conference on Neural Networks*, pages 1942–1948, 1995.

Udayan Khurana, Deepak Turaga, Horst Samulowitz, and Srinivasan Parthasrathy. Cognito: Automated Feature Engineering for Supervised Learning. In *IEEE International Conference on Data Mining*, pages 1304–1307, 2016.

Udayan Khurana, Horst Samulowitz, and Deepak Turaga. Feature Engineering for Predictive Modeling Using Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, pages 3407–3414, 2018a.

Udayan Khurana, Horst Samulowitz, and Deepak Turaga. Ensembles with Automated Feature Engineering. In *International Conference on Machine Learning AutoML Workshop*, 2018b.

Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. *CoRR*, abs/1605.0, 2016.

Aaron Klein, Stefan Falkner, Numair Mansur, and Frank Hutter. RoBO: A Flexible and Robust Bayesian Optimization Framework in Python. In *NIPS Bayesian Optimization Workshop*, 2017.

Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune: A Derivative-free Optimization Framework for Hyperparameter Tuning. In *ACM International Conference on Knowledge Discovery and Data Mining*, pages 443–452, 2018.

Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. In *International Conference on Machine Learning AutoML Workshop*, pages 2825–2830, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *International Conference on Neural Information Processing Systems*, volume 1, pages 1097–1105, 2012.

Alexandre Lacoste, Hugo Larochelle, Mario Marchand, and François Laviolette. Sequential Model-Based Ensemble Optimization. In *arXiv preprint arXiv:1402.0796*, 2014.

Manuel Laguna and Rafael Martí. Experimental Testing of Advanced Scatter Search Designs for Global Optimization of Multimodal Functions. *Journal of Global Optimization*, 33 (2):235–255, 2005.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40: 1–58, 2017.

Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327*, 2017.

Scott Langevin, David Jonker, Christopher Bethune, Glen Coppersmith, Casey Hilland, Jonathon Morgan, Paul Azunre, and Justin Gawrilow. Distil: A Mixed-Initiative Model Discovery System for Subject Matter Experts. In *International Conference on Machine Learning AutoML Workshop*, 2018.

John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits. In *Advances in Neural Information Processing Systems*, pages 817–824, 2008.

Steven M. LaValle, Michael S. Branicky, and Stephen R. Lindemann. On the Relationship Between Classical Grid Search and Probabilistic Roadmaps. *The International Journal of Robotics Research*, 23:673–692, 2004.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *arXiv preprint arXiv:1603.06560*, 2016.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18:1–52, 2018.

Marius Lindauer and Frank Hutter. Warmstarting of Model-based Algorithm Configuration. In *AAAI Conference on Artificial Intelligence*, pages 1355–1362, 2018.

Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

Gang 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):1–15, 2016.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based Hyperparameter Optimization through Reversible Learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

Hunter McGushion. HyperparameterHunter, 2019. URL `https://github.com/HunterMcGushion/hyperparameter_hunter`.

Ines Ben Messaoud, Haikal El Abed, Volker Märgner, and Hamid Amiri. A design of a preprocessing framework for large database of historical documents. In *Workshop on Historical Document Imaging and Processing*, pages 177–183, 2011.

Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of Bayesian methods for seeking the extremum. *Towards Global Optimisation*, 2:117–129, 1978.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107:1495–1515, 2018.

Marcin Molga and Czesław Smutnicki. Test functions for optimization needs. *Test functions for optimization needs*, 2005.

Fatemeh Nargesian, Horst Samulowitz, Udayan Khurana, Elias B. Khalil, and Deepak Turaga. Learning Feature Engineering for Classification. In *International Joint Conference on Artificial Intelligence*, pages 2529–2535, 2017.

Thomas Nickson, Michael A. Osborne, Steven Reece, and Stephen Roberts. Automated Machine Learning on Big Data using Stochastic Algorithm Tuning. *CoRR*, 2014.

Randal S. Olson and Jason H. Moore. TPOT : A Tree-based Pipeline Optimization Tool for Automating Machine Learning. In *International Conference on Machine Learning AutoML Workshop*, pages 66–74, 2016.

Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Genetic and Evolutionary Computation Conference*, pages 485–492, 2016a.

Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. Automating biomedical data science through tree-based pipeline optimization. In *Applications of Evolutionary Computation*, pages 123–137. Springer International Publishing, 2016b.

Preston Parry. auto_ml, 2019. URL `https://github.com/ClimbsRocks/auto_ml`.

Konstantinos E. Parsopoulos. Particle Swarm Methods. In *Handbook of Heuristics*, pages 1–47. Springer International Publishing, 2016.

Emanuel Parzen. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1961.

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. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Fabian Pedregosa. Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, pages 737–746, 2016.

Riccardo Poli, William B. Langdon, Nicholas Freitag McPhee, and John R. Koza. *A Field Guide to Genetic Programing*. Lulu.com, 2008.

Gil Press. Data Scientists Spend Most of Their Time Cleaning Data, 2016. URL `https://whatsthebigdata.com/2016/05/01/data-scientists-spend-most-of-their-time-cleaning-data/`.

Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *Journal of Machine Learning Research*, 20:1–32, 2019.

Pavel Pudil, Jana Novovičová, and Josef Kittler. Floating search methods in feature selection. *Pattern recognition letters*, 15(11):1119–1125, 1994.

Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, Inc., 1999.

Erhard Rahm and Hong Hai Do. Data cleaning: Problems and Current Approaches. In *IEEE Data Engineering Bulletin*, 2000.

Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the Gini Index and Information Gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1): 77–93, 2004.

Vijayshankar Raman and Joseph M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *International Conference on Very Large Data Bases*, volume 1, pages 381–390, 2001.

RapidMiner. Introducing RapidMiner Auto Model, 2018. URL `https://rapidminer.com/resource/automated-machine-learning/`.

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

Matthias Reif, Faisal Shafait, and Andreas Dengel. Meta-learning for evolutionary parameter optimization of classifier. *Machine Learning*, 87:357–380, 2012.

Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*, 21(4):25–34, 1987.

Herbert Robbins. Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are Loss Functions All the Same? *Neural Computation*, 16(5):1063–1076, 2004.

Reuven Y. Rubinstein and Dirk P. Kroese. *Simulation and the Monte Carlo Method*. Wiley, 2008.

Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, 2007.

Manuel Martin Salvador, Marcin Budka, and Bogdan Gabrys. Towards automatic composition of multicomponent predictive systems. In *International Conference on Hybrid Artificial Intelligence Systems*, pages 27–39, 2016.

B. Samanta. Gear fault detection using artificial neural networks and support vector machines with genetic algorithms. *Mechanical Systems and Signal Processing*, 18(3):625–644, 2004.

Brandon Schoenfeld, Christophe Giraud-Carrier, Mason Poggemann, Jarom Christensen, and Kevin Seppi. Preprocessor Selection for Machine Learning Pipelines. In *International Conference on Machine Learning AutoML Workshop*, 2018.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*, pages 1–19, 2017.

Micah J. Smith, Roy Wedge, and Kalyan Veeramachaneni. FeatureHub: Towards collaborative data science. In *IEEE International Conference on Data Science and Advanced Analytics*, pages 590–600, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

Jan A. Snyman. *Practical Mathematical Optimization: An introduction to basic optimization theory and classical and new gradient-based algorithms*. Springer, 2005.

Francisco J. Solis and Roger J.-B. Wets. Minimization By Random Search Techniques. *Mathematics of Operations Research*, 6(1):19–30, 1981.

Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating model search for large scale machine learning. In *ACM Symposium on Cloud Computing*, pages 368–380, 2015.

Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian Optimization with Robust Bayesian Neural Networks. In *Advances in Neural Information Processing Systems*, pages 4134–4142, 2016.

Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. ATM: A distributed, collaborative, scalable system for automated machine learning. In *IEEE International Conference on Big Data*, pages 151–162, 2017.

Kevin Swersky, Jasper Snoek, and Ryan P. Adams. Freeze-Thaw Bayesian Optimization. *arXiv preprint arXiv:1406.3896*, 2014.

Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *ACM International Conference on Knowledge Discovery and Data Mining*, pages 847–855, 2013.

USU Software AG. Katana, 2018. URL `https://katana.usu.de/`.

Jan N. van Rijn and Frank Hutter. Hyperparameter Importance Across Datasets. In *International Conference on Knowledge Discovery and Data Mining*, pages 2367–2376, 2018.

Jan N. van Rijn, Salisu Mamman Abdulrahman, Pavel Brazdil, and Joaquin Vanschoren. Fast Algorithm Selection Using Learning Curves. In *International Symposium on Intelligent Data Analysis*, 2015.

Joaquin Vanschoren. Meta-learning. In *Automatic Machine Learning: Methods, Systems, Challenges*, pages 39–68. Springer, 2018a.

Joaquin Vanschoren. Meta-Learning: A Survey. *CoRR*, abs/1810.0:1–29, 2018b.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: networked science in machine learning. *ACM International Conference on Knowledge Discovery and Data Mining*, 15(2):49–60, 2014.

Zhou Wang and Alan C. Bovik. Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures. *IEEE Signal Processing Magazine*, 26(1):98–117, 2009.

Gellert Weisz, Andras Gyorgy, and Csaba Szepesvari. LeapsAndBounds: A Method for Approximately Optimal Algorithm Configuration. In *International Conference on Machine Learning AutoML Workshop*, pages 5257–5265, 2018.

Marcel Wever, Felix Mohr, and Eyke Hüllermeier. ML-Plan for Unlimited-Length Machine Learning Pipelines. In *International Conference on Machine Learning AutoML Workshop*, 2018.

Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Hyperparameter Search Space Pruning - A New Component for Sequential Model-Based Hyperparameter Optimization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 104–119, 2015a.

Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Learning Hyperparameter Optimization Initializations. In *IEEE International Conference on Data Science and Advanced Analytics*, 2015b.

Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Automatic Frankensteining: Creating Complex Ensembles Autonomously. In *SIAM International Conference on Data Mining*, pages 741–749, 2017.

David H. Wolpert. Stacked Generalization. *Neural Networks*, 5(2):241–259, 1992.

David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

Yiming Yang and Jan O. Pedersen. A Comparative Study on Feature Selection in Text Categorization. *International Conference on Machine Learning*, 97:412–420, 1997.

Zhi-Qiang Zeng, Hong-Bin Yu, Hua-Rong Xu, Yan-Qi Xie, and Ji Gao. Fast training Support Vector Machines using parallel sequential minimal optimization. In *International Conference on Intelligent System and Knowledge Engineering*, pages 997–1001, 2008.

Linda Zhou. How to Build a Better Machine Learning Pipeline, 2018. URL `https://www.datanami.com/2018/09/05/how-to-build-a-better-machine-learning-pipeline/`.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, volume 23, pages 2595–2603, 2010.

Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *CoRR*, abs/1611.0:1–16, 2017.