Benchmark and Survey of Automated Machine Learning Frameworks

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 (ML) 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. This paper is a combination of a survey on current AutoML methods and a benchmark of popular AutoML frameworks on real data sets. Driven by the selected frameworks for evaluation, we summarize and review important AutoML techniques and methods concerning every step in building an ML pipeline. The selected AutoML frameworks are evaluated on 137 different data sets.

1. Introduction

In recent years 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 and 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 variable 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., 2018b).

This paper is a combination of a short survey on AutoML and an evaluation of popular frameworks for AutoML and HPO on real data. We select in total 16 different AutoML and HPO frameworks for evaluation. The different techniques used by those frameworks are summarized to provide an overview for the reader. This way, research concerning the automation of any aspect of an ML pipeline is reviewed: determining the pipeline shape, selecting an ML algorithm for each stage in a pipeline and tuning each algorithm.

This paper focuses on classical machine learning and does **not** consider neural network architecture search while still many of the ideas can be transferred. Most topics discussed in this survey are large enough to be handled in dedicated surveys. Consequently, this paper does not aim to handle each topic in exhaustive depth but aims to provide a profound overview.

The contributions of this paper are as following:

- We introduce a mathematical formulation covering the complete procedure of automatic ML pipeline creation. Existing problem formulations, e.g., Escalante et al. (2009); Bergstra et al. (2011); Thornton et al. (2013); Hutter et al. (2018a), are extended to also include pipeline structure search.
- We review open-source frameworks for automatically building an ML pipeline.
- An empirical evaluation of various HPO algorithms on 137 real data is conducted. To the best of our knowledge, this is the first independent benchmark of HPO algorithms.
- An empirical evaluation of various AutoML frameworks on 73 real data is conducted. To the best of our knowledge, this is the most extensive evaluation—in terms of tested algorithms as well as used data sets—of AutoML frameworks.

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.

Lately, several surveys regarding AutoML have been published. Elshawi et al. (2019) and He et al. (2019) focus on automatic neural network architecture search—which is not covered in this survey—and only briefly introduce methods for classic machine learning. Quanming et al. (2018) and Hutter et al. (2018a) cover less steps of the pipeline creation process and do not provide an empirical evaluation of the presented methods. Finally, Tuggener et al. (2019) provides only a high-level overview. For a more detailed survey, the reader is conferred to one of those surveys. Two benchmarks of AutoML methods have been published so far. Balaji and Allen (2018) and Gijsbers et al. (2019) evaluate various AutoML frameworks on real data sets. Our evaluations exceed those benchmarks in terms of evaluated data sets as well as evaluated frameworks. Both benchmarks focus only on a performance comparison while we also take a look at the obtained ML models and pipelines. Furthermore, both benchmarks do not consider HPO methods.

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 introduces the evaluated AutoML frameworks. The evaluation is presented 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. Let a fixed set of basic algorithms, e.g., various classification, imputation and feature selection algorithms, be given as $\mathcal{A} = \left\{A^{(1)}, A^{(2)}, \ldots, A^{(n)}\right\}$. Each algorithm $A^{(i)}$ is configured by a vector of hyperparameters $\vec{\lambda}^{(i)}$ from the domain $\Lambda^{(i)}$.

Without loss of generality let a pipeline structure be modeled as an directed acyclic graph (DAG). Each node represents a basic algorithm. The edges represent the flow of an input data set through the different algorithms. Often the DAG shape is restricted by implicit constraints, i.e., a pipeline for a classification problem has to have a classification algorithm as the last step. Let G denote the set of valid pipeline structures and $|g|$ denote the length of a pipeline, i.e., the number of nodes in g, for $g \in G$.

Definition 1 (Machine Learning Pipeline) *Let a triplet $(g, \vec{A}, \vec{\lambda})$ define an ML pipeline with $g \in G$ a valid pipeline shape, $\vec{A} \in \mathcal{A}^{|g|}$ a vector consisting of the selected algorithm for each node and $\vec{\lambda}$ a vector comprising the hyperparameters of all selected algorithms. The pipeline is denoted as $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$.*

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. Let a data set be defined as $D = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$. A pipeline trained on D is denoted as $\mathcal{P}_{g,\vec{A},\vec{\lambda},D}$. Given a trained pipeline it is important to assess its performance in order to build pipelines with a low generalization error.

Definition 2 (Pipeline Performance) *Let a trained pipeline $\mathcal{P}_{g,\vec{A},\vec{\lambda},D}$ be given. Given a data set D' of size m and a loss metric $\mathcal{L}(\cdot,\cdot)$, the performance π of $\mathcal{P}_{g,\vec{A},\vec{\lambda},D}$ is calculated as*

$$\pi\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},D}, D'\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}_i, y_i),$$

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

Let an *ML task* be defined by a given data set, loss function and an 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. Finally, for each selected algorithm its corresponding hyperparameters have to be selected. 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 $\Lambda^{(\cdot)}$ and a set of valid pipeline structures G be given. Furthermore, let a data set D be given. Then, the pipeline creation problem consists of finding a pipeline structure together with a joint algorithm and hyperparameter selection that minimizes the loss*

$$g^\star, \vec{A}^\star, \vec{\lambda}^\star \in \underset{g\in G, \vec{A}\in\mathcal{A}^{|g|}, \vec{\lambda}\in\Lambda}{\arg\min}\ \pi\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},D}, D\right). \tag{1}$$

To limit the effects of overfitting, Equation (1) is often augmented by cross-validation. Let the data set D be split into K folds $\{D^{(1)}_{\text{valid}}, \ldots, D^{(K)}_{\text{valid}}\}$ and $\{D^{(1)}_{\text{train}}, \ldots, D^{(K)}_{\text{train}}\}$ such that $D^{(i)}_{\text{train}} = D \setminus D^{(i)}_{\text{valid}}$. The final objective function is defined as

$$g^\star, \vec{A}^\star, \vec{\lambda}^\star \in \underset{g\in G, \vec{A}\in\mathcal{A}^{|g|}, \vec{\lambda}\in\Lambda}{\arg\min}\ \frac{1}{K}\sum_{i=1}^{K}\pi\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},D^{(i)}_{\text{train}}}, D^{(i)}_{\text{valid}}\right).$$

Using Equation (1), the pipeline creation problem is formulated as a black box optimization problem. Finding the global optimum in such equations 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. 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).

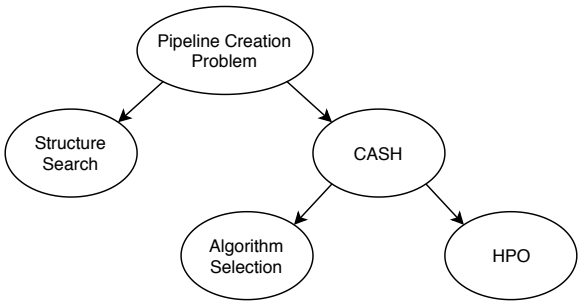

Figure 1: Subproblems of the pipeline creation problem.

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 is sufficient. In contrast, most current state-of-the-art algorithms solve the pipeline creation problem in a single step. Figure 1 shows a schematic representation of the different optimization problems for the automatic composition 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. 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. This stage highly depends on the underlying domain. Finally, a single model is trained on the previously selected features.

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

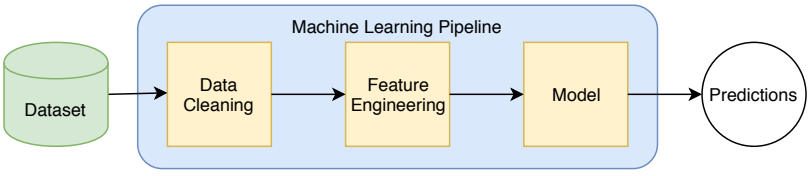

Figure 2: Prototypical ML pipeline. First, the input data is cleaned; next, features are extracted. Finally, the transformed input is passed through an ML model to create predictions.

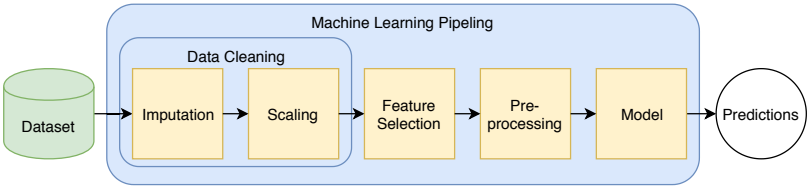

Figure 3: Fixed ML pipeline used by most AutoML frameworks. Minor differences exist regarding the implemented data cleaning steps.

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 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. Often single steps in this pipeline can be omitted.

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.

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 (Koza, 1992; Banzhaf et al., 1997). Genetic programming has been used for automatic program code

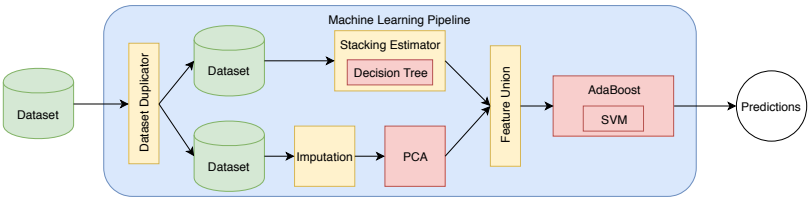

Figure 4: Specialized ML pipeline for a specific ML task.

generation for a long time (Poli et al., 2008). Yet, the application to pipeline structure selection is quite recent. Pipelines are interpreted as tree structures that are generated via genetic programming. Two individuals are combined by selecting sub-graphs of the pipeline structures and combining these sub-graphs to a new graph. Mutation can be implemented by random addition or deleting of a node. This way flexible tree-shaped pipelines can be generated.

Hierarchical task networks (HTNs) (Ghallab et al., 2004) 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 obtained. 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).

Monte-Carlo tree search (Kocsis and Szepesvári, 2006; Browne et al., 2012) is a heuristic best-first tree search algorithm. Similar to hierarchical planning, ML pipelines structure generation is reduced to finding the best node in the search tree. However, instead of decomposing complex tasks pipelines with increasing complexity are iteratively created (Rakotoarison et al., 2019).

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 in combination with Monte-Carlo tree search is used to select a pipeline $\mathcal{P}^{(i)}$ by predicting its performance $\pi(\mathcal{P}^{(i)})$ and probabilities which action to chose in this state (Drori et al., 2018).

4. Algorithm Selection and Hyperparameter Optimization

Let a shape $g \in G$, a loss function $\mathcal{L}$ and a training set D 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). For readability, assume $|g| = 1$. The CASH problem is defined as

$$\vec{A}^\star, \vec{\lambda}^\star \in \underset{\vec{A}\in\mathcal{A},\vec{\lambda}\in\Lambda}{\arg\min}\ \pi\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},D}, D\right).$$

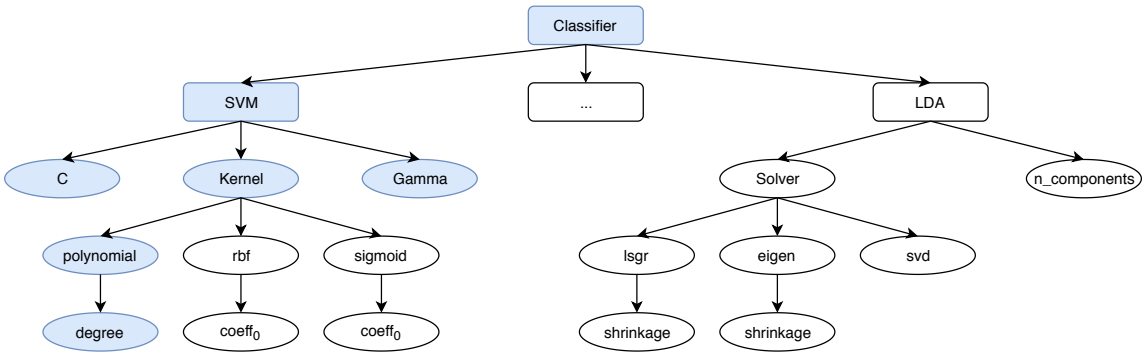

Figure 5: Incomplete representation of the structured configuration space for selecting and tuning a classification algorithm. Rectangle nodes represent the selection of an algorithm. Ellipse nodes represent tunable hyperparameters. Highlighted in blue is an active configuration to select and configure a SVM with a polynomial kernel.

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

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

referred to as the *configuration space*. This leads to the final CASH minimization problem

$$\vec{\lambda}^{\star} \in \arg\min_{\vec{\lambda} \in \Lambda} \; \pi \left(\mathcal{P}_{g,\vec{\lambda},D}, D \right). \tag{2}$$

This definition can be easily extended for $|g| > 1$ by introducing a distinct λ_r for each node. For readability, let $f(\vec{\lambda}) = \pi \left(\mathcal{P}_{g,\vec{\lambda},D}, D \right)$ be denoted as the *objective function*.

It is important to note that Equation (2) is not easily solvable as the search space is quite large and complex. As hyperparameters can be categorical and real-valued, Equation (2) is a mixed-integer nonlinear optimization problem (Belotti et al., 2013). Furthermore, conditional dependencies between different hyperparameters exist. 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 $\vec{\lambda} \in \Lambda^{(i)}$ can be aggregated in two groups: mandatory hyperparameters always have to be present while conditional hyperparameters depend on the selected value of another hyperparameter. 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 5. 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.

The rest of this section introduces different optimization strategies to solve Equation (2).

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. Even though grid search is easily implemented and parallelized (Bergstra and Bengio, 2012), it has two major drawbacks: 1) it 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) and 2) the hierarchical hyperparameter structure is not considered, leading to redundant configurations.

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. Random search is straightforward to implement and parallelize and well suited for gradient-free functions with many local minima (Solis and Wets, 1981). 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 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 so-far 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 Figure 6.

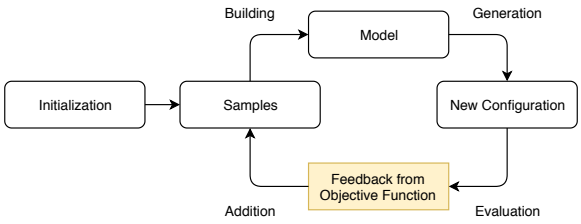

Figure 6: Schematic procedure of SMBO. During the initialization a set of configuration and score tuples is created. These samples are used to create a regression model of the objective function. Next, a new configuration is selected and evaluated by the objective function. Finally, the new tuple is added to the set of samples.

The loss function $f(\vec{\lambda})$ is complemented by a probabilistic regression model M that acts as a surrogate for f. The surrogate model M allows predicting the performance of an arbitrary configuration $\vec{\lambda}$ without evaluating the demanding objective function. M is built using all so-far observed performances $D_{1:n} = \left\{ \left(\vec{\lambda}_1, f(\vec{\lambda}_1)\right), \ldots, \left(\vec{\lambda}_n, f(\vec{\lambda}_n)\right) \right\}$ and is used to sequentially create new configurations. These new configurations are obtained using a cheap acquisition function. Each proposed configuration is evaluated on the objective function f and the result added to $D_{1:n}$. These steps are repeated until a fixed budget T—usually either a fixed number of iterations or a time limit—is exhausted. The initialization is often implemented by selecting a small number of random configurations.

Even though fitting a model and selecting a configuration introduces a 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.

To actually implement the surrogate model fitting and configuration selection, 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

$$P\left(f \mid D_{1:n}\right) \propto P\left(D_{1:n} \mid f\right) P(f). \tag{3}$$

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 $\vec{\lambda}_{n+1} \in \Lambda$ 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).

The surrogate model M corresponds to the posterior in Equation (3). 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 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. A common drawback of Gaussian processes is the runtime complexity of $\mathcal{O}(n^3)$(Rasmussen and Williams, 2006). However, as long as multi-fidelity methods (see Section 7) are not used, this is not relevant for AutoML as evaluating a high number of configurations is prohibitively expensive. A more relevant drawback for CASH is the missing native support of categorical input[1] and utilization of the search space structure.

Random forest regression (Breiman, 2001) is an ensemble method consisting of multiple regression trees (Breiman et al., 1984). Regression trees use recursive splitting of the training data to create groups of similar observations. Besides the ability to natively handle categorical variables, random forests are fast to train and even faster on evaluating new data while obtaining a good predictive power.

1. Extensions for treating integer variables in Gaussian processes exist, e.g., (Garrido-Merchán and Hernández-Lobato, 2017).

In contrast to the two previous surrogate models, a tree-structured Parzen estimator (TPE) (Bergstra et al., 2011) does not model the posterior $p(f \mid D_{1:n})$ directly. Instead the likelihood $p(D_{1:n} \mid f)$ is modeled. Using a performance threshold f', all observed configurations are split into two sets: one for well performing configurations and one for bad performing configurations. Using kernel density estimation (KDE) (Parzen, 1961), those sets are transformed into two distributions. Regarding the tree structure, TPE natively handles hierarchical search spaces by modeling each hyperparameter individually. These distributions are connected hierarchically representing the dependencies between the hyperparameters resulting in a pseudo multidimensional distribution.

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. In general, evolutionary algorithms are applicable to a wide variety of optimization problems as no assumptions about the objective function are necessary.

Escalante et al. (2009) and Claesen et al. (2014) perform hyperparameter optimization using 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). 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.

4.5 Multi-Armed Bandit Learning

Many SMBO methods suffer from the mixed and conditional search space. By performing grid search considering only the categorical hyperparameters, the configuration space can be split into a finite set of smaller configuration spaces—called a *hyperpartition*—containing only continuous hyperparameters. Each hyperpartition can be optimized by standard Bayesian optimization methods. The selection of a hyperpartition can be modeled as a *multi-armed bandit problem* (Robbins, 1952). Even though multi-armed bandit learning can also be applied to continuous optimization (Munos, 2014), in the context of AutoML it is only used in a finite setting in combination with other optimization techniques (Hoffman et al., 2014; Efimova et al., 2017; Gustafson, 2018; das Dôres et al., 2018).

4.6 Gradient Descent

A very powerful optimization method is *gradient descent*, an iterative minimization algorithm. 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 and therefore gradient descent in general not applicable. By assuming some properties of f—and therefore limiting the applicability of this approach to specific problem instance—gradient descent can still be used (Maclaurin et al., 2015; Pedregosa, 2016). Due to the rigid constraints, gradient descent is not analyzed in more detail.

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, redundant entries, invalid values or broken links between entries of multiple data sets (Rahm and Do, 2000). In general, data cleaning is split in two tasks: error detection and error repairing (Chu et al., 2016). For over one decade semi-automatic, interactive systems exist to aid a data scientist in data cleaning (Galhardas et al., 2000; Raman and Hellerstein, 2001). Yet, most current approaches still aim to assist a human data scientist instead of fully automate data cleaning, e.g., (Krishnan et al., 2015; Khayyaty et al., 2015; Krishnan et al., 2016; Eduardo and Sutton, 2016; Rekatsinas et al., 2017). Krishnan and Wu (2019) proposed a semi-automatic data cleaning procedure: based on a human defined *data quality* function, data cleaning is treated similar to pipeline structure search. Basic data cleaning operators are iteratively combined using greedy search to create sophisticated data cleaning.

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. In general, current AutoML frameworks do not consider state-of-the-art data cleaning research.

Sometimes, high requirements for specific data qualities are introduced by later stages in an ML pipeline, e.g., SVMs require a numerical encoding of categorical features while random forests can handle them natively. 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).

Incorporating domain knowledge during data cleaning heavily increases the data quality (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). Feature engineering can be split in three sub-tasks: feature extraction, feature construction and feature selection (Motoda and Liu, 2002). Feature engineering—especially feature creation—is highly domain specific and very difficult to generalize. Even for a data

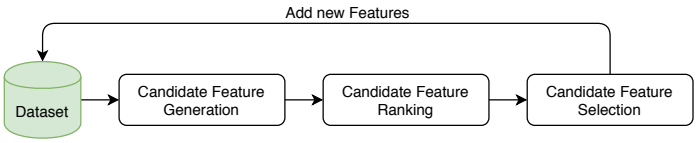

Figure 7: Iterative feature generation procedure.

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. In the context of AutoML feature extraction and feature construction are usually aggregated as feature generation.

6.1 Feature Generation

Feature generation creates new features through a functional mapping of the original features (feature extraction) or discovering missing relationships between the original features (feature creation) (Motoda and Liu, 2002). In general, this step requires the most domain knowledge and is therefore the hardest to automate. Approaches to enhance automatic feature generation with domain knowledge, e.g., (Friedman and Markovitch, 2015; Smith et al., 2017), are not considered as AutoML aims to be domain-agnostic. Still, some features—like dates or addresses—can be easily transformed without domain knowledge to extract more meaningful features (Chen et al., 2018).

Basically all automatic feature generation approaches follow the iterative scheme displayed in Figure 7. 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.

New features are generated using a predefined set of operators transforming the original features (Sondhi, 2009):

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. Using correlation tests and regression models, the correlation between two features can be expressed as a new feature (Kaul et al., 2017).

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.

Similar to pipeline structure search, feature generation can be considered as a node selection problem in a *transformation tree*: 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).

Many approaches augment feature selection with an ML model to actually calculate the performance of the new feature set: Early approaches combined beam search in combination

with different heuristics to explore the feature space in a best-first way (Markovitch and Rosenstein, 2002). More recently, greedy search (Dor and Reich, 2012; Khurana et al., 2016) and depth-first search (Lam et al., 2017) in combination with feature selection have been used to create a sequence of operators. In each iteration, a random operation is applied to the currently best-performing data set until the performance improvement does converge. Another popular approach is combining features using genetic programming (Smith and Bull, 2005; Tran et al., 2016).

Instead of iteratively exploring the transformation tree, exhaustive approaches consider a fully expanded transformation tree up to a predefined depth (Kanter and Veeramachaneni, 2015; Katz et al., 2017). Most of the candidate features do not contain meaningful information. Consequently, the set of candidate features has to be filtered. Yet, generating exponentially many features makes this approach prohibitively expensive in combination with an ML model. Instead the new features can be filtered without an actual evaluation (see Section 6.2) or ranked based on meta-features (see Section 7.5). Based on the meta-features of a candidate feature the expected loss reduction after including this candidate can be predicted using a regression model (Katz et al., 2017; Nargesian et al., 2017), reinforcement learning (Khurana et al., 2018a) or stability selection (Kaul et al., 2017). The predictive 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.

Some frameworks specialize on feature generation in relational databases (Kanter and Veeramachaneni, 2015; Lam et al., 2017). Chen et al. (2018) proposed using stacked estimators. The predicted output is added as an additional feature such that later estimators can correct wrongly labeled data. Finally, Khurana et al. (2018b) proposed to create an ensemble of sub-optimal feature sets (see Section 7.4).

6.2 Feature Selection

Feature selection chooses a subset of the original feature set to speed up the subsequent ML model training and improve its performance by removing redundant or misleading features (Motoda and Liu, 2002). Furthermore, the interpretability of the trained model is increased. A simple domain-agnostic filtering approach for feature selection is based on information theory and statistics (Pudil et al., 1994; Yang and Pedersen, 1997; Dash and Liu, 1997; Guyon and Elisseeff, 2003). Algorithms like univariate selection, variance threshold, feature importance, correlation matrices (Saeys et al., 2007) or stability selection (Meinshausen and Bühlmann, 2010) 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) and selected via standard CASH methods. More advanced feature selection methods are usually implemented in dedicated feature engineering frameworks.

In general, the feature set—and consequently also its power set—is finite. Feature selection via *wrapper functions* searches for the best feature subset by testing its performance on a specific ML algorithm. Simple approaches use random search or test the power set exhaustively (Dash and Liu, 1997). Heuristic approaches follow an iterative procedure by adding single features (Kononenko, 1994). Margaritis (2009) used a combination of forward and backward selection to select a feature-subset while Gaudel and Sebag (2010) proposed to model the subset selection as a reinforcement problem. Vafaie and De Jong (1992) used

genetic programming in combination with a cheap prediction algorithm to obtain a well performing feature subset.

Finally, special feature selection methods exist that are useful in combination with feature extraction and feature creation. Tran et al. (2016) used genetic programming to construct new features. In addition, the information how often each feature was used during feature construction is re-used to obtain a feature importance. Katz et al. (2017) proposed to calculate meta-features for each new feature, e.g., diversity of values or mutual information with the other features. Using a pre-trained classifier, the influence of a single feature can be predicted to select only promising features.

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 (Fernández-Godino et al., 2016). 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 (1) by introducing an additional budget term $s \in [0, 1]$ that 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 8, 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 to evaluate each remaining configuration. This procedure is repeated until only one configuration remains (Hutter et al., 2018b). 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. It calculates the number of configurations and budget size based on some budget constraints. A descending sequence of configuration numbers

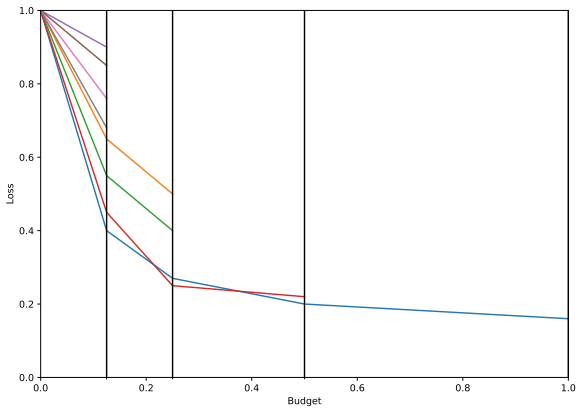

Figure 8: Schematic representation of SuccessiveHalving with eight different configurations.

m is calculated and passed to SuccessiveHalving. Consequently, no prior knowledge is required anymore for SuccessiveHalving.

Fabolas uses multi-objective optimization instead of deterministically calculated budgets to reduce model loss and training time. Therefore, a Gaussian process is trained on the combined input $(\vec{\lambda}, s)$. Additionally the acquisition function is enhanced by entropy search (Hennig and Schuler, 2012). This allows predicting the performance of $\vec{\lambda}_i$, tested with budget s_i, for the full budget $s = 1$.

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 unpromising 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 (Maron and Moore, 1993; Hutter et al., 2011).

The training of an ML model is often an iterative procedure converging to a local minimum. By observing the improvement in each iteration, the learning curve of an ML model can be predicted (Swersky et al., 2014; Domhan et al., 2015; Klein et al., 2017b). By simultaneously considering multiple configurations in an iterative procedure at once, the most promising configuration can be optimized in each step.

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 (Weisz et al., 2018).

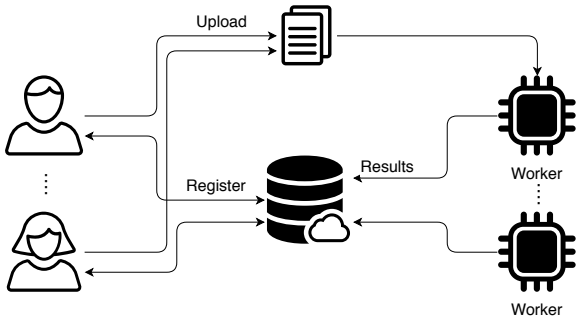

Figure 9: Components of an AutoML service (Swearingen et al., 2017).

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). 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. 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. Alternatively, an uncompleted evaluation of a configuration can be modeled as the worst possible result (Kandasamy et al., 2017). This way, new configurations can be sampled from an approximated posterior while preventing evaluating the same configuration twice.

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 9. 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).

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 (Khurana et al., 2018b; 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).

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 (2018) provides an survey exclusively on meta-learning.

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 assess the importance of single hyperparameters allowing to remove unimportant hyperparameters from the configuration space (Hutter et al., 2014; Wistuba et al., 2015a; van Rijn and Hutter, 2018; Probst et al., 2019) or identify promising regions (Wistuba et al., 2015b).

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 based on the predicted performance e.g., (Alia and Smith-Miles, 2006; Wistuba et al., 2015a; Nargesian et al., 2017) or ranking of the models, e.g., (Sohn, 1999; Gama and Brazdil, 2000). Consequently, the risk of superfluous configuration evaluations is minimized. The same techniques can also be used to directly create promising configurations in the first place.

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). (Gomes et al., 2012)

Pipeline Structure Meta-learning is also applicable for pipeline structure search. Using information which preprocessing 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

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

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 $\vec{\lambda}_i$ and calculated performance $\pi(t_j, \vec{\lambda}_i)$. Given a new task t_{new}, a meta-learner L is trained on $\mathbf{P}$ to recommend a configuration $\vec{\lambda}_{\text{new}}$.

A simple, task-independent approach for ranking configurations is sorting $\mathbf{P}$ by the performance. Configurations with a higher performance are more favorable (Vanschoren, 2018). 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 $\vec{m}(t_{\text{new}})$, 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, 2018).

8. Selected Frameworks

This section provides an introduction to the evaluated AutoML frameworks. Frameworks were selected based on their popularity, namely the number of citations and GitHub stars. Furthermore, all frameworks should use different techniques and had to be open source.

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 properties is available in Table 1.

Algorithm	Solver	Λ	Parallel	Time.	Cat.
Dummy	–	no	no	no	no
Random Forest	–	no	no	no	no
Grid Search	Grid Search	no	Local	no	yes
Random Search	Random Search	no	Local	no	yes
RoBO	SMBO with Gaussian process	no	no	no	no
BTB	Bandit learning and Gaus. process	yes	no	no	yes
hyperopt	SMBO with TPE	yes	Cluster	no	yes
SMAC	SMBO with random forest	yes	Local	yes	yes
BOHB	Bandit learning and TPE	yes	Cluster	yes	yes
Optunity	Particle Swarm Optimization	yes	Local	no	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 (Parallel), whether a timeout for a single evaluation exists (Time.) and if categorical variables are natively supported (Cat.).

8.1.1 Baseline Methods

To assess the effectiveness of the different CASH algorithms, two baseline methods are added: a dummy classifier and a random forest. The dummy classifier uses stratified sampling to create random predictions. For both methods the scikit-learn (Pedregosa et al., 2011) implementations with default hyperparameters are used.

8.1.2 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 evaluations of all available ML algorithms while also reducing the search space significantly by eliminating redundant configurations. When all grid search instances have finished, the best result of all instances is returned.

8.1.3 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) implemen-

tation 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.

8.1.4 RoBO

RoBO (Klein et al., 2017a) 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). In the context of this work, RoBO is configured to use SMBO with a Gaussian process as a surrogate model. The hyperparameters of the Gaussian process are automatically tuned using Markov chain Monte Carlo sampling.

A major limitation of RoBO is the missing support of categorical hyperparameters. In combination with the missing support for conditional hyperparameters RoBO is rather unsuited for CASH as many unnecessary fitting procedures with inactive parameters are executed. RoBO does not support any performance improvements except Fabolas (see Section 7.1). RoBO is evaluated in version 0.3.1.

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. BTB provides multiple policies for selecting a hyperpartition but in the context of this work upper confidence bound is used. The remaining continuous hyperparameters are selected using Bayesian optimization with Gaussian processes similar to RoBO. The acquisition function samples random configurations and orders them by their expected improvement. 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. BTB is evaluated in version 0.2.5.

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. The number of iterations is only limited in number and not in elapsed time.

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). hyperopt is evaluated in version 0.2.

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. Additionally, new configurations are randomly sampled from the complete configuration space.

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. SMAC is evaluated in version 0.10.0.

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.

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. BOHB is evaluated in version 0.7.4.

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 Particle swarm optimization (PSO) solver is used. Optunity supports a structured configuration space similar to HYPEROPT. 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. Optunity is evaluated in version 1.0.0.

8.2 AutoML Frameworks

This section presents the selected frameworks for AutoML. All presented frameworks are capable of building a complete ML pipeline based on the methods provided in Sections 3,

Framework	CASH Solver	Structure	Ensem.	Cat. In.	Parallel	Time.
Dummy	–	Fixed	no	no	no	no
Random Forest	–	Fixed	no	no	no	no
TPOT	Genetic Prog.	Variable	no	no	Local	yes
hpsklearn	hyperopt	Fixed	no	yes	no	yes
auto-sklearn	SMAC	Fixed	yes	Enc.	Cluster	yes
Random Search	Random Search	Fixed	no	Enc.	Cluster	yes
ATM	BTB	Fixed	no	yes	Cluster	no
H2O AutoML	Grid Search	Fixed	yes	yes	Cluster	yes

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

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 Baseline Methods

To assess the effectiveness of the different AutoML algorithms, two baseline methods are added: 1) a dummy classifier using stratified sampling to create random predictions and 2) a simple pipeline consisting of an imputation of missing values and a random forest. For both baseline methods the scikit-learn (Pedregosa et al., 2011) implementation is used.

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 performance 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.

TPOT supports basically all popular scikit-learn preprocessing, classification and regression methods. It is evaluated in version 0.10.2.

8.2.3 Hyperopt-Sklearn

hyperopt-sklearn or hpsklearn (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 based on 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.

hyperopt-sklearn supports only a very limited data preprocessing, namely principal component analysis (PCA), standard or min-max scaling and normalization. One hot encoding and string preprocessing are disabled by default. Additionally, the most popular scikit-learn classification and regression methods are supported. hyperopt-sklearn is evaluated in version 0.0.3.

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 optional categorical encoding, imputation, removing variables with low variance and optional scaling—is executed. Next, an optional preprocessing and mandatory modeling algorithm are selected and tuned via SMAC. To process categorical data, a manual label encoding of the data set in combination with explicitly listing the categorical features is necessary. As the name already implies, auto-sklearn uses scikit-learn for all ML algorithms. The sister package Auto-WEKA (Thornton et al., 2013; Kotthoff et al., 2016) provides very similar functionality for the WEKA library.

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 best pipelines is created. auto-sklearn is evaluated in version 0.5.2.

8.2.5 Random Search

Random search is added as additional baseline method with tuned hyperparameters. It is based on auto-sklearn. Instead of using SMAC, configurations are generated randomly. Additionally, ensemble building and meta-learning are disabled.

8.2.6 ATM

ATM (Swearingen et al., 2017) is a collaborative service to build optimized classification pipelines. This framework has a strong emphasis on parallelization allowing the distribution of single evaluations in a cluster. Currently, ATM uses a simple pipeline structure with an optional PCA, an optional standard or min-max scaling followed by a tunable classification algorithm. All algorithms are based on scikit-learn and popular classification algorithms are supported. 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. ATM is evaluated in version 0.2.2.

8.2.7 H2O AutoML

H2O (H2O.ai, 2019) is a distributed ML framework to assist data scientists. It aims to support a data scientist in every aspect of daily work. In the context of this paper only the H2O AutoML component is considered. H2O AutoML is able to automatically select and tune a classification algorithm without preprocessing. Configurations are generated using Cartesian or random grid search in combination with an overall runtime budget. In the end, the best performing configurations are aggregated to create an ensemble. Besides the open-source AutoML version, H2O also provides a commercial AutoML solution called DriverlessAI that is not considered.

In contrast to all other evaluated frameworks, H2O is developed in Java with Python bindings and does not use scikit-learn. H2O is evaluated in version 3.26.0.8.

9. Experiments

This section provides empirical evaluations of different CASH and pipeline building algorithms. At first, the comparability of the results is discussed and the methodology of the benchmarks is explained. Next, the usage of synthetic data sets is shortly evaluated. Finally, all selected frameworks are empirically evaluated on real data.

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 papers 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 distributions, 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. 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 9.3) and real data sets (see Section 9.5) 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.

Recently, an open-source AutoML benchmark has been published by Gijsbers et al. (2019). By integrating AutoML frameworks via simple adapters, a fair comparison under standardized conditions is possible. Currently only four different AutoML frameworks and no CASH algorithms at all are integrated. Yet, this approach is very promising to provide an empirical basis for AutoML in the future.

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

9.2 Benchmarking Methodology

All experiments were conducted using *n1-standard-8* virtual machines from Google Cloud Platform equipped with Intel Xeon E5 processors with 8 cores and 30 GB memory[3]. Each virtual machine used Ubuntu 18.04.02, Python 3.6.7 and scikit-learn 0.21.3. To eliminate the effects of non-determinism, all experiments are repeated ten times with different random seeds and results are averaged. Three different types of experiments with different setups were conducted:

1. Synthetic test functions (see Section 9.3) are limited to exactly 250 iterations. The performance is defined as the minimal absolute distance

$$\min_{\vec{\lambda}_i \in \Lambda} |f(\vec{\lambda}_i) - f(\vec{\lambda}^*)|$$

   between the considered configurations $\vec{\lambda}_i$ and the global optimum $\vec{\lambda}^*$.

2. CASH solvers (see Section 9.5.1) are limited to exactly 325 iterations. Preliminary evaluations have shown that all algorithm basically always converge before hitting this iteration limit. The model fitting in each iteration is limited to a cut-off time of ten minutes. Configurations violating this time limit are assigned the worst possible performance. The performance of each configuration is determined using a 4-fold cross-validation. As loss function, the accuracy

$$\mathcal{L}_{\text{Acc}}(\hat{y}, y) = \frac{1}{|y|} \sum_{i=1}^{|y|} \mathbb{1}(\hat{y}_i = y_i) \tag{4}$$

   is used, with $\mathbb{1}$ being an indicator function.

3. AutoML frameworks (see Section 9.5.2) are limited by a soft-limit of 1 hour and a hard-limit of 1.25 hours. Fitting of single configurations is aborted after ten minutes if the framework supports a cut-off time. The performance of each configuration is determined using a 4-fold cross-validation in combination with the accuracy as loss function.

The evaluation timeout of ten minutes 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. Preliminary tests revealed, that all algorithms are limited by CPU power and not available memory. Therefore, the memory consumption is not further considered. All frameworks supporting parallelization are configured to used eight threads.

For the third experiment we also preliminary tested cut-off timeouts of 4 and 8 hours on ten random data sets. The performance after 4 or even 8 hours did only marginally improve in comparison to 1 hour.

The source code used for the benchmarks is available online[4].

3. For more information see `https://cloud.google.com/compute/docs/machine-types`.
4. Available at `https://github.com/Ennosigaeon/automl_benchmark`.

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.

9.3 Synthetic Test Functions

A common strategy applied for many years is using synthetic test functions for benchmarking, e.g., (Snoek et al., 2012; Eggensperger et al., 2015; Klein et al., 2017a). Due to the closed-form representation, the synthetic loss for a given configuration can be computed in constant time.

All CASH algorithms from Section 8 are tested on various synthetic test functions. Grid search and random search are used as base line algorithms. Table 3 contains the performance of each algorithm after the completed optimization. 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.

Synthetic test functions do not allow a simulation of categorical hyperparameters leading to an unrealistic, completely unstructured configuration space. 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. Finally, all synthetic test functions have a continuous and smooth surface. These properties do not hold for real response surfaces (Eggensperger et al., 2015). This implies that synthetic test functions are not suited for CASH benchmarking.

9.4 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 algorithm 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. 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 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. 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 used directly.

9.5 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 benchmark.

As explained in Section 2, the performance of an AutoML algorithm depends on the tested data set. Consequently, it is not useful to evaluate the 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, a combination of the curated benchmarking suites OpenML100[5] (Bischl et al., 2017), OpenML-CC18[6] (Bischl et al., 2019) and AutoML Benchmark[7] (Gijsbers et al., 2019) is used. The combination of these benchmarking suits contains 137 classification tasks with high-quality data sets having between 500 and $600,000$ samples and less than $7,500$ features. However, high-quality does not imply that no preprocessing of the data is necessary as for example some data sets contain missing values. A complete list of all evaluated data sets with some basic meta-features is provided in Appendix A. No CASH algorithm and most AutoML frameworks do not support categorical features. Therefore, categorical features of all data sets are transformed using one hot encoding.

5. Available at `https://www.openml.org/s/14`.
6. Available at `https://www.openml.org/s/99`.
7. Available at `https://www.openml.org/s/218`.

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.

9.5.1 CASH Algorithms

At first, all previously mentioned CASH algorithms are tested on all data sets. Therefore, a hierarchical configuration space containing 13 classifiers with a 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. Some algorithms 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.

For grid search 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.

Table 9 in Appendix C contains the raw results of the evaluation. Reported are the average accuracy over all trials per data set. 23 of the evaluated data sets contain missing values. As no algorithm in the configuration space is able to handle missing values, all evaluations on these data sets failed and are not further considered.

In the following, accuracy scores are normalized to obtain data set independent evaluations. Therefore, the accuracy per data set is normalized across all evaluated algorithms to an interval between zero and one. Zero represents the performance of the dummy classifier

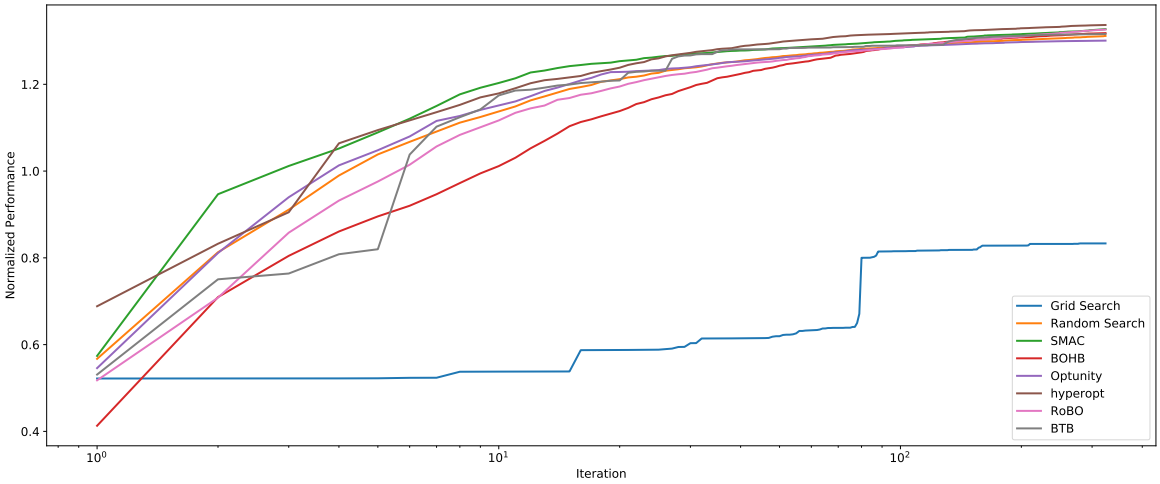

Figure 10: Normalized performance of the incumbent per iteration. Results are averaged over all data sets and 10 repetitions.

and one the performance of the random forest. Algorithms outperforming the random forest baseline obtain results greater than one.

Figure 10 shows the performance of the best incumbent per iteration averaged over all data sets. It is important to note that the results for the very first iterations are slightly skewed due to the parallel evaluation of candidate configurations. Iterations are recorded in order of finished evaluation timestamps, meaning that 8 configurations started in parallel are recorded as 8 distinct iterations.

It is apparent that all methods except grid search are able to outperform the random forest baseline within roughly 10 iterations. After 325 iterations, all algorithms converge to similar performance measures. The individual performances after 325 iterations are also displayed in Figure 11.

A pair-wise comparison of the performances of the final incumbent is displayed in Figure 12. The comparison with grid search is omitted due to spacial constrictions. Figure 12 reveals that—with very few exceptions—all algorithm obtain similar precision scores across all data sets.

Figure 13 shows the raw scores for each CASH framework over 10 repetitions for 40 data sets. Those data sets were selected as they show the highest deviation of the scores over the 10 repetitions. The remaining data sets yielded very consistent results, similar to the *steel-plates-fa* data set. We do not know which data set properties are responsible for the unstable results.

Finally, we examine the similarity of the proposed configurations per data set. Therefore, each configuration is transformed to a numerical representation. Numerical hyperparameters are normalized by their according search space, categorical hyperparameters are not transformed. We decided to only compare configurations with each other having the same classification algorithm. For each classification algorithm, all configuration vectors are aggregated using mean shift clustering (Fukunaga and Hostetler, 1975) with a bandwidth $h = 0.25$. To account for the mixed-type vector representations, the Gower distance (Gower,

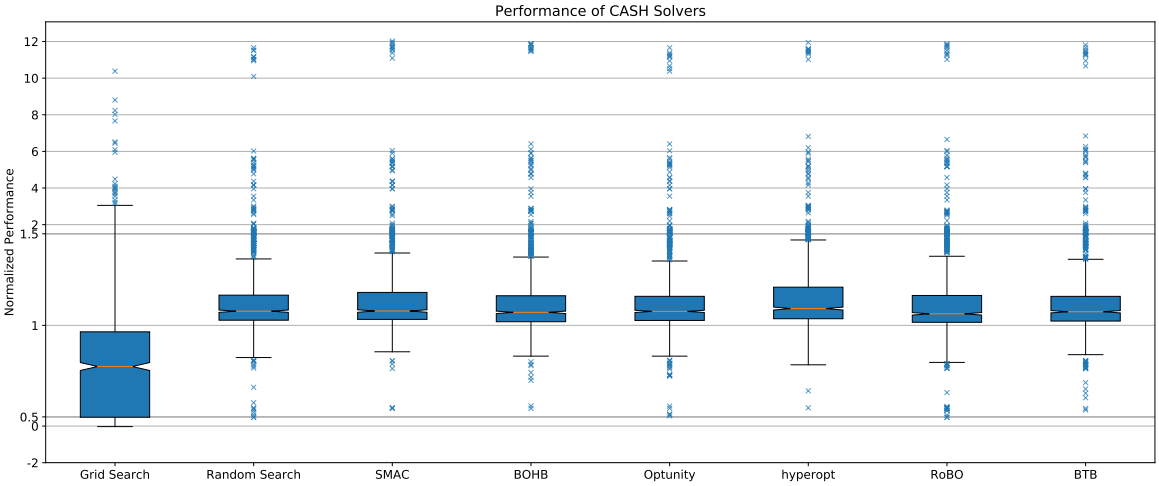

Figure 11: Normalized performance of the final incumbent per CASH solvers. For better readability, performances between 0.5 and 1.5 are stretched out.

1971) is used as the distance metric between two configurations. To assess the quality of the resulting clusters—and therefore also the overall configuration similarity—, the silhouette coefficient (Rousseeuw, 1987) is computed.

Figure 14 shows the silhouette coefficient versus number of instances per cluster. Displayed are clusters of all configurations and per CASH algorithm. On average, each CASH algorithm yields 3.0670 ± 2.3772 different classification algorithms. Most clusters contain only a few configurations with a low silhouette coefficient indicating that the resulting hyperparameters have a high variance.

We require clusters to contain at least 5 configurations to be considered as similar. In addition, the silhouette coefficient has to be greater than 0.75. In total 106 of 114 data sets contain at least one cluster with similar configurations. However, most of those clusters are created by grid search which usually yields identical configurations for each trial. 11 data sets yield configurations with a high similarity for at least half of the CASH algorithms. However, for most data sets configurations are very dissimilar. It is not apparent which meta-features are responsible for those results. In summary, most CASH procedures yield highly different hyperparameters on most data sets depending on the random seed.

9.5.2 AutoML Frameworks

Finally, AutoML frameworks capable of building complete ML pipelines are evaluated. Therefore, all data sets from the AutoML Benchmark suite are used. Additionally, all data sets from the OpenML100 suite containing missing values and all data sets in the OpenML-CC18 not contained in OpenML100 are selected. The final list of all 73 selected data sets is provided in Table 10 in Appendix C.

ATM does not provide the possibility to abort configuration evaluations after a fixed time. Therefore, ATM often exceeds the total time budget. To ensure the time budget, all configuration evaluations are manually aborted after 1.25 hours. All parameters are left at the default value. auto-sklearn is the only framework supporting a memory limitation.

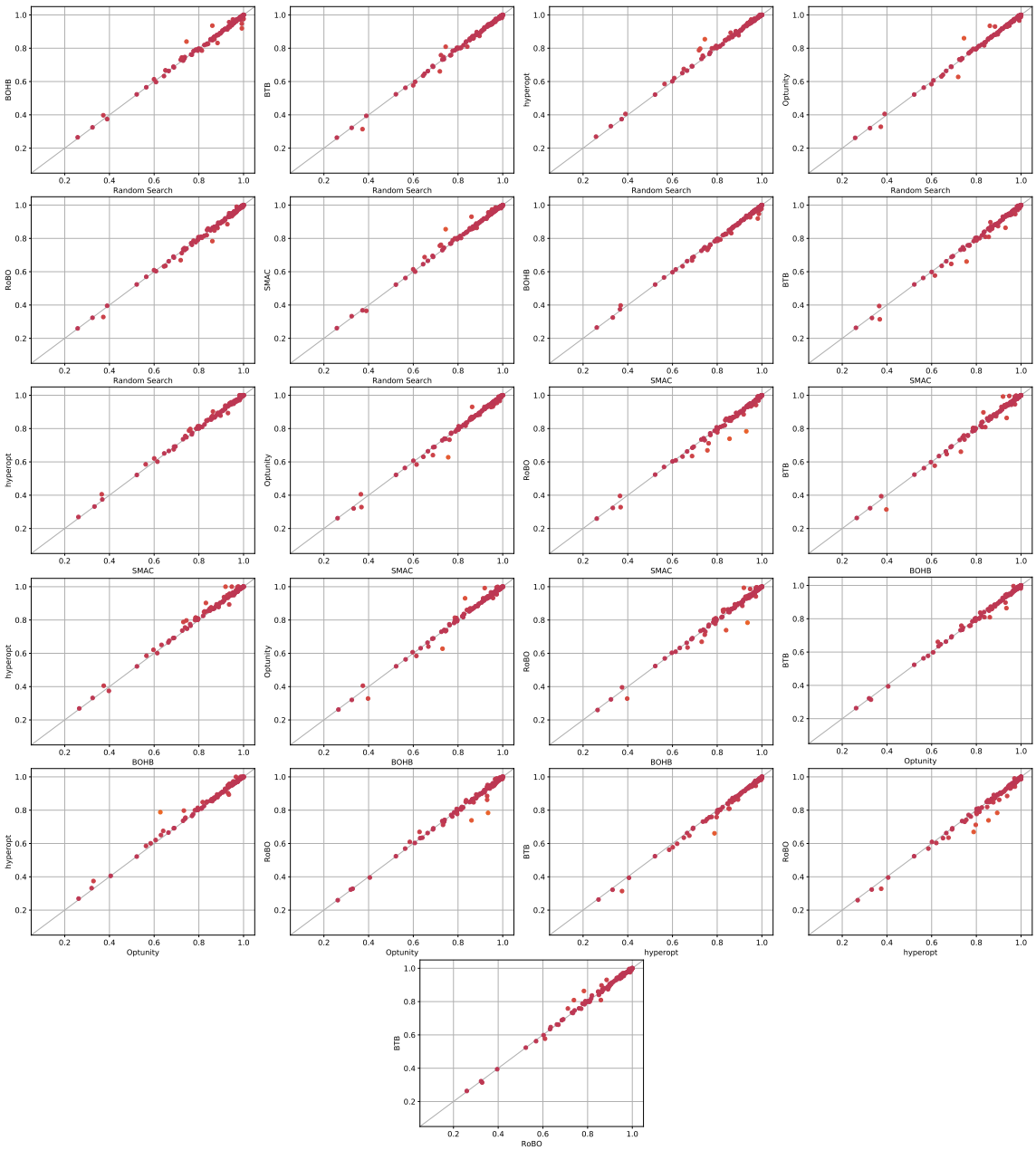

Figure 12: Pair-wise comparison of the mean precision of CASH algorithms. The axes represent the accuracy score of the stated CASH algorithm. Each point represents the averaged results for a single data set.

Figure 13: Raw and averaged accuracy of all CASH solvers on selected data sets.

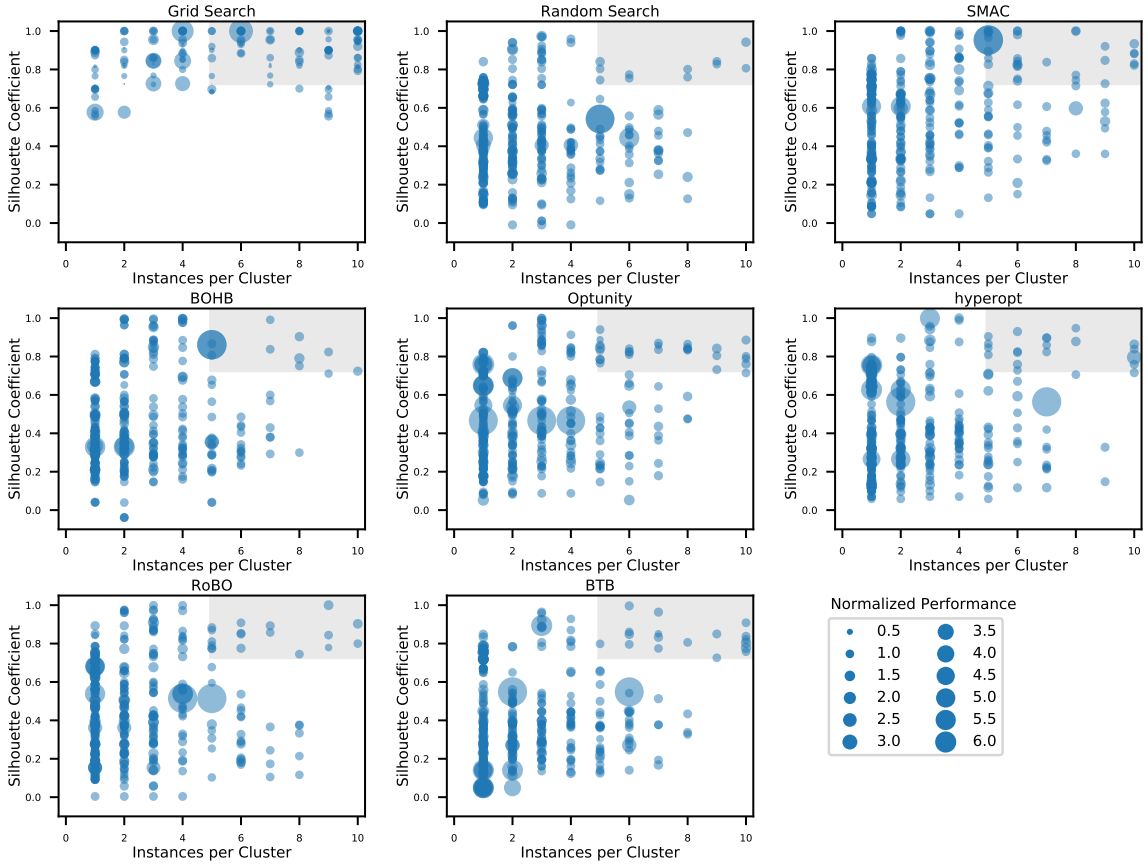

Figure 14: Similarity of configurations versus number of instances per cluster. Each marker represents the similarity of configurations for a single data set and single classification algorithm. The marker size indicates the normalized accuracy (larger equals higher accuracy). Clusters in the highlighted area are considered to contain similar configurations. Each subplot considers only configurations yielded by the stated CASH algorithm.

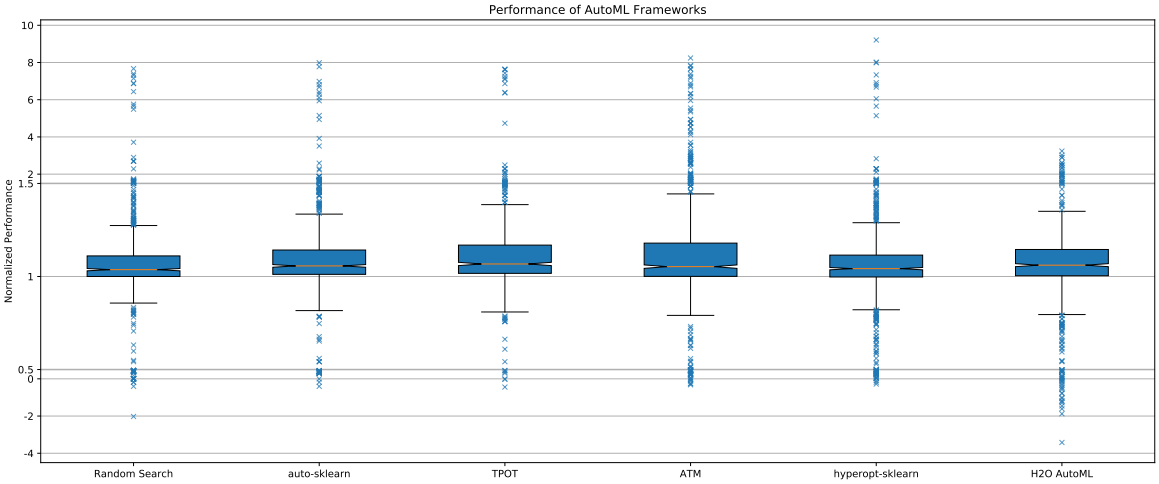

Figure 15: Normalized performance of the final pipeline per AutoML framework. For better readability, performances between 0.5 and 1.5 are stretched out.

The memory limit is set to 4096MB per thread to fully utilize the available memory. All other parameters are left at the default value. Random Search uses auto-sklearn with a random configuration generation. Again, the memory limit is set to 4096MB per thread to fully utilize the available memory. Meta-learning and ensemble support are deactivated. All other parameters are left at the default value. hyperopt-sklearn does not support the parallel evaluation of multiple configurations. Consequently, only single-threaded evaluation of configurations is used. Furthermore, hyperopt-sklearn was manually extended to support a time budget instead of number of iterations. To suggest new configuration, TPE is used. All other parameters are left at the default value. TPOT, Random Forest, H2O AutoML and the stratified dummy classifier are used with their default parameters.

Table 10 in Appendix C contains the raw results of the evaluation. Reported are the average accuracy over all trials per data set. In contrast to the CASH algorithms, the AutoML frameworks struggled with various data sets. ATM drops samples in training data sets with missing values. Data sets 38, 1111, 1112, 1114 and 23380 contain missing values for every single sample. Consequently, ATM uses an empty training set and crashes. hyperopt-sklearn is very fragile, especially regarding missing values. If the very first configuration evaluation of a data set fails, hyperopt-sklearn aborts the optimization. To compensate this issue the very first evaluation is repeated upto 100 times. Furthermore, the optimization often does not stop after the soft-timeout for no apparent reason. TPOT sometimes crashed with a segmentation fault. For multiple data sets TPOT was stopped in the first generation. Consequently, only random search without genetic programming was performed. Data sets 40923, 41165 and 41167 consistently timed out. auto-sklearn and random search both violated the memory constraints on the data sets 40927, 41159 and 41167. Finally, for H2O AutoML the Java server consistently crashed for no apparent reason on the data sets 40978, 41165, 41167 and 41169. Data set 41167 is the largest evaluated data set. This could explain why so many frameworks are struggling with this specific data set. In the following analysis these failing data sets are ignored.

Algorithm	TPOT	hpsklearn	auto-sklearn	Random	ATM	H2O
TPOT	0.1190	0.1106	0.0379	0.0356	0.0519	0.1165
hpsklearn	0.1106	0.1926	0.0517	0.0461	0.0828	0.1414
auto-sklearn	0.0379	0.0517	0.5996	0.5542	0.0557	0.0202
Rand. Search	0.0356	0.0461	0.5542	0.5307	0.0329	0.0266
ATM	0.0519	0.0828	0.0557	0.0329	0.4591	0.0
H2O	0.1165	0.1414	0.0202	0.0266	0.0	0.3135

Table 5: Averaged pair-wise Levenshtein ratio on original ML pipelines.

Figure 15 contains the normalized performances of all AutoML frameworks averaged over all data sets. It is apparent, that all frameworks are able to outperform the random forest baseline on average. However, single results vary significantly. The pair-wise comparison in Figure 16 shows that all frameworks yield pipelines with similar performances on average.

Figure 17 shows raw scores for each AutoML framework over 10 trials for 40 data sets. Those data sets were selected as they show the highest deviation of the scores over the 10 trials. About 50% of the evaluated data sets show a high variance in the obtained results. The remaining data sets yield very consistent performances. It is not clear which data set features are responsible for this separation.

Finally, Figure 18 provides an overview of often constructed pipelines. For readability, pipelines were required to be constructed at least thrice to be included in the graph. Ensembles of pipelines are treated as distinct pipelines.

TPOT, ATM, hyperopt-sklearn and H2O AutoML produce on average pipelines with less than two steps. Consequently, the cluster of pipelines around the root node is created by those AutoML frameworks. Basically all pipelines in the left and right subgraph were created by the two auto-sklearn variants. To further assess the similarity of the resulting ML pipelines, we transform each pipeline to a string by mapping each algorithm to a distinct letter. The similarity between two pipelines is than expressed by the Levenshtein ratio (Levenshtein, 1966; Ratcliff and Metzener, 1988). Table 5 shows the averaged pair-wise Levenshtein ratio of all pipelines per AutoML framework.

It is apparent, that random search and auto-sklearn have a high similarity with each other and them self. This can be explained by the long (semi-)fixed pipeline structure. All other AutoML frameworks yield very low similarity ratios. This can partially be explained by the different search spaces, i.e., the AutoML frameworks do not support identical base algorithms. Therefore, we also consider an abstract representation of the ML pipelines, e.g., replacing all classification algorithms with an identical symbol. Table 6 shows that TPOT, hyperopt-sklearn, ATM and H2O build similar pipelines. auto-sklearn and random search build pipelines that differ strongly from the remaining frameworks but are still very similar to each other.

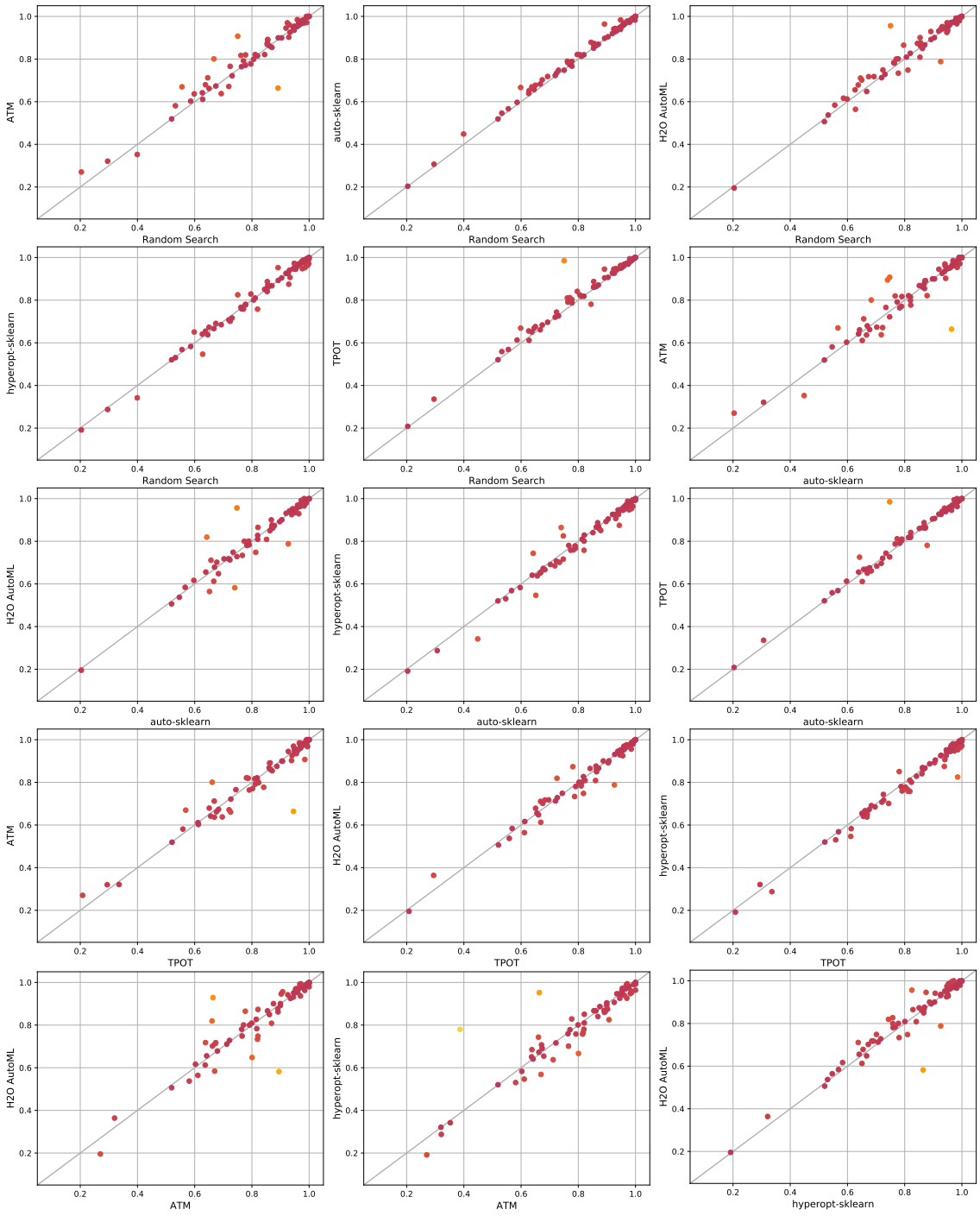

Figure 16: Pair-wise comparison of normalized performances of AutoML frameworks. The axes represent the accuracy score of the stated AutoML framework. Each point represents the averaged results for a single data set.

Figure 17: Raw and averaged accuracy of all AutoML frameworks on selected data sets.

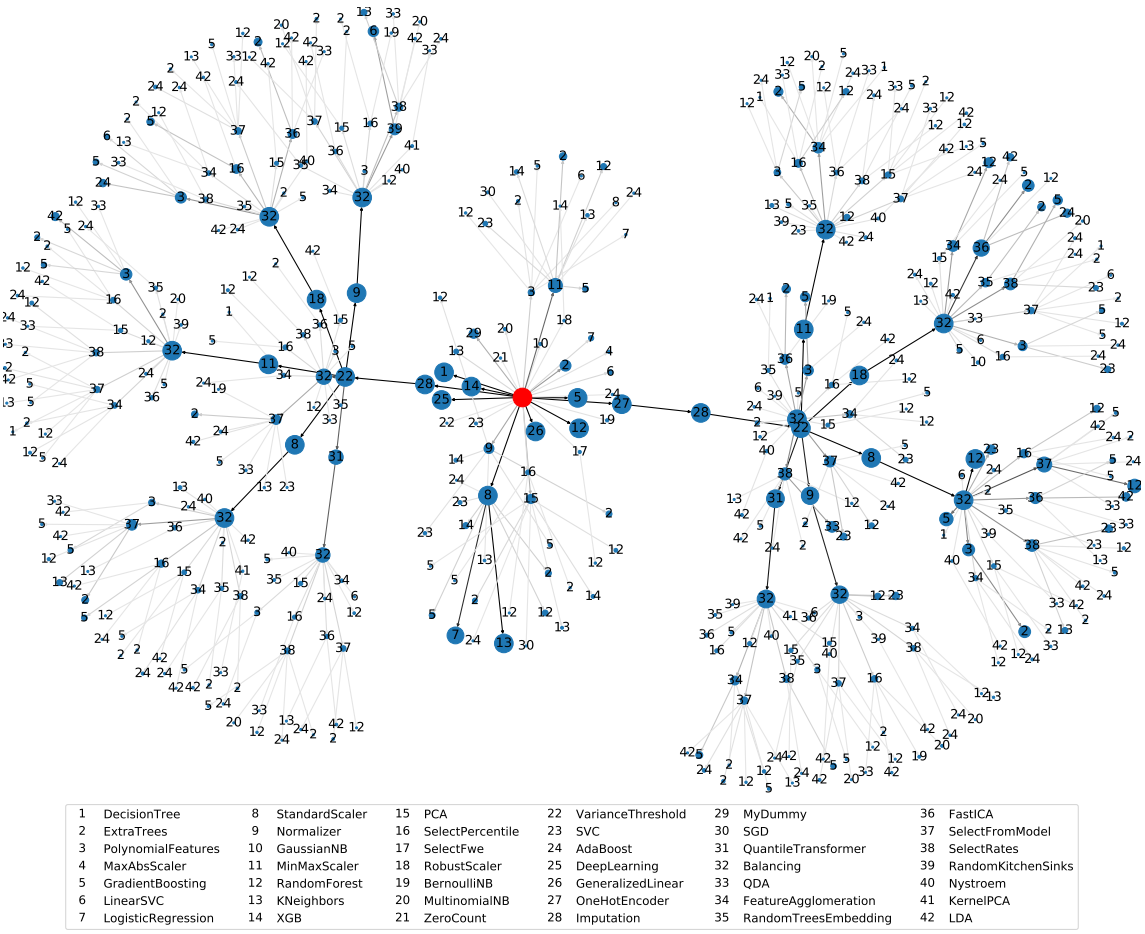

Figure 18: Overview of constructed ML pipelines. The node size and edge color indicate the popularity of specific (sub-)pipelines. The red node represents the root node. Pipelines are created by following the graph from the root to a leaf node.

Algorithm	TPOT	hpsklearn	auto-sklearn	Random	ATM	H2O
TPOT	0.7784	0.7330	0.3300	0.3674	0.7234	0.8595
hpsklearn	0.7330	0.7995	0.4048	0.4377	0.8208	0.7877
auto-sklearn	0.3300	0.4048	0.9104	0.8790	0.4164	0.2803
Rand. Search	0.3674	0.4377	0.8790	0.8423	0.4490	0.3272
ATM	0.7234	0.8208	0.4164	0.4490	0.8524	0.7769
H2O	0.8595	0.7877	0.2803	0.3272	0.7769	1.0

Table 6: Averaged pair-wise Levenshtein ratio on generalized ML pipelines.

10. Discussion and Opportunities for Future Research

The experiments in Section 9.5.1 revealed that all CASH algorithms, except grid search, perform on average very similar. Surprisingly, random search did not perform worse than the other algorithms. The performance of the final configurations differs only marginally; for most data sets the absolute differences are less than 1% accuracy. Consequently, a ranking of CASH algorithms on pure performance measures is not reasonable. In future, other aspects like scalability or method overhead should also be considered.

On average all AutoML frameworks performed quite similar. However, for single data sets performances differ on average by 6% accuracy leading to more unstable results. In addition, the CASH algorithms performed better than the AutoML frameworks on nearly all data sets. This is again a surprising result as each CASH algorithm spends on average only 12 minutes optimizing a single data in contrast to the 1 hour of AutoML frameworks. Possible explanations could be the significantly larger search spaces of AutoML frameworks or a smaller number of evaluated configurations due to internal overhead, e.g., cross-validations. Further evaluations are necessary to explain this behaviour.

Currently, AutoML frameworks build pipelines with an average length of less than 2.5 components. This is partly caused by frameworks with a short, fixed pipeline layout. Yet, also TPOT yields pipelines with less than 1.5 components on average. Consequently, the potential of specialized pipelines is currently not utilized at all. A benchmarking of other frameworks capable of building flexible pipelines, e.g., ML-Plan (Mohr et al., 2018; Wever et al., 2018), P4ML (Gil et al., 2018) or MOSAIC (Rakotoarison et al., 2019), in combination with longer optimization periods is desirable to better understand the capabilities of creating adaptable pipelines.

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. 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 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 in the context of AutoML. Regarding feature creation, most methods (naively) combine predefined operators with features. For building flexible pipelines currently only a few 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).

The experiments revealed that some data sets are better suited for AutoML than others. Currently, we can not explain which data set meta-features are responsible for this behavior. A better understanding of the relation between data set meta-features and AutoML algorithms may enable AutoML for the failing data sets and boost meta-learning.

Following the CRISP-DM (Shearer, 2000), AutoML currently focuses only the modeling stage. However, to successfully conduct an ML project all stages in the CRISP-DM should be considered. To make AutoML truly available for novice users, integration of data acquisition and deployment measures are necessary. In general, AutoML currently does not consider lifecycle management at all.

11. Conclusion

In this paper, we have provided a theoretical and empirical introduction to the current state of AutoML. We provided the first empirical evaluation of CASH algorithms on 114 publicly available real-world data sets. Furthermore, we conducted the largest evaluation of AutoML frameworks in terms of considered frameworks as well as number of data sets. Important techniques used by those frameworks are theoretically introduced and summarized. This way, we presented the most important research for automating each step of creating an ML pipeline. Finally, we extended current problem formulations to cover the complete process of building ML pipelines.

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 still very basic and 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.

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. Evaluated Data Sets

Data Set		Classes	Samples	Numeric Feat.	Categorical Feat.	Missing Values	Incom. Samples	Minority %
kr-vs-kp	(3)	2	3196	0	37	0	0	47.78
letter	(6)	26	20000	16	1	0	0	3.67
balance-scale	(11)	3	625	4	1	0	0	7.84
mfeat-factors	(12)	10	2000	216	1	0	0	10.00
mfeat-fourier	(14)	10	2000	76	1	0	0	10.00
breast-w	(15)	2	699	9	1	16	16	34.48
mfeat-karhunen	(16)	10	2000	64	1	0	0	10.00
mfeat-morpholog	(18)	10	2000	6	1	0	0	10.00
mfeat-pixel	(20)	10	2000	0	241	0	0	10.00
car	(21)	4	1728	0	7	0	0	3.76
mfeat-zernike	(22)	10	2000	47	1	0	0	10.00
cmc	(23)	3	1473	2	8	0	0	22.61
mushroom	(24)	2	8124	0	23	2480	2480	48.20
optdigits	(28)	10	5620	64	1	0	0	9.86
credit-approval	(29)	2	690	6	10	67	37	44.49
credit-g	(31)	2	1000	7	14	0	0	30.00
pendigits	(32)	10	10992	16	1	0	0	9.60
segment	(36)	7	2310	19	1	0	0	14.29
diabetes	(37)	2	768	8	1	0	0	34.90
sick	(38)	2	3772	7	23	6064	3772	6.12
soybean	(42)	19	683	0	36	2337	121	1.17
spambase	(44)	2	4601	57	1	0	0	39.40
splice	(46)	3	3190	0	61	0	0	24.04
tic-tac-toe	(50)	2	958	0	10	0	0	34.66
vehicle	(54)	4	846	18	1	0	0	23.52
waveform-5000	(60)	3	5000	40	1	0	0	33.06
electricity	(151)	2	45312	7	2	0	0	42.45
satimage	(182)	6	6430	36	1	0	0	9.72

eucalyptus	(188)	5	736	14	6	448	95	14.27
isolet	(300)	26	7797	617	1	0	0	3.82
vowel	(307)	11	990	10	3	0	0	9.09
scene	(312)	2	2407	294	6	0	0	17.91
monks-problems-	(333)	2	556	0	7	0	0	50.00
monks-problems-	(334)	2	601	0	7	0	0	34.28
monks-problems-	(335)	2	554	0	7	0	0	48.01
JapaneseVowels	(375)	9	9961	14	1	0	0	7.85
synthetic_contr	(377)	6	600	60	2	0	0	16.67
irish	(451)	2	500	2	4	32	32	44.40
analcatdata_aut	(458)	4	841	70	1	0	0	6.54
analcatdata_dmf	(469)	6	797	0	5	0	0	15.43
profb	(470)	2	672	5	5	1200	666	33.33
collins	(478)	15	500	20	4	0	0	1.20
mnist_784	(554)	10	70000	784	1	0	0	9.02
sylva_agnostic	(1036)	2	14395	216	1	0	0	6.15
gina_agnostic	(1038)	2	3468	970	1	0	0	49.16
ada_agnostic	(1043)	2	4562	48	1	0	0	24.81
mozilla4	(1046)	2	15545	5	1	0	0	32.86
pc4	(1049)	2	1458	37	1	0	0	12.21
pc3	(1050)	2	1563	37	1	0	0	10.24
jm1	(1053)	2	10885	21	1	25	5	19.35
kc2	(1063)	2	522	21	1	0	0	20.50
kc1	(1067)	2	2109	21	1	0	0	15.46
pc1	(1068)	2	1109	21	1	0	0	6.94
KDDCup09_appete	(1111)	2	50000	192	39	8024152	50000	1.78
KDDCup09_churn	(1112)	2	50000	192	39	8024152	50000	7.34
KDDCup09_upsell	(1114)	2	50000	192	39	8024152	50000	7.36
MagicTelescope	(1120)	2	19020	11	1	0	0	35.16
airlines	(1169)	2	539383	3	5	0	0	44.54
artificial-char	(1459)	10	10218	7	1	0	0	5.87
bank-marketing	(1461)	2	45211	7	10	0	0	11.70
banknote-authen	(1462)	2	1372	4	1	0	0	44.46
blood-transfusi	(1464)	2	748	4	1	0	0	23.80
cardiotocograph	(1466)	10	2126	35	1	0	0	2.49
climate-model-s	(1467)	2	540	20	1	0	0	8.52
cnae-9	(1468)	9	1080	856	1	0	0	11.11
eeg-eye-state	(1471)	2	14980	14	1	0	0	44.88
first-order-the	(1475)	6	6118	51	1	0	0	7.94
gas-drift	(1476)	6	13910	128	1	0	0	11.80
har	(1478)	6	10299	561	1	0	0	13.65
hill-valley	(1479)	2	1212	100	1	0	0	50.00
ilpd	(1480)	2	583	9	2	0	0	28.64
madelon	(1485)	2	2600	500	1	0	0	50.00
nomao	(1486)	2	34465	89	30	0	0	28.56
ozone-level-8hr	(1487)	2	2534	72	1	0	0	6.31
phoneme	(1489)	2	5404	5	1	0	0	29.35
one-hundred-pla	(1491)	100	1600	64	1	0	0	1.00
one-hundred-pla	(1492)	100	1600	64	1	0	0	1.00
one-hundred-pla	(1493)	100	1599	64	1	0	0	0.94
qsar-biodeg	(1494)	2	1055	41	1	0	0	33.74
wall-robot-navi	(1497)	4	5456	24	1	0	0	6.01
semeion	(1501)	10	1593	256	1	0	0	9.73
steel-plates-fa	(1504)	2	1941	33	1	0	0	34.67
tamilnadu-elect	(1505)	20	45781	2	2	0	0	3.05
wdbc	(1510)	2	569	30	1	0	0	37.26
micro-mass	(1515)	20	571	1300	1	0	0	1.93
wilt	(1570)	2	4839	5	1	0	0	5.39
adult	(1590)	2	48842	6	9	6465	3620	23.93
covertype	(1596)	7	581012	10	45	0	0	0.47
Bioresponse	(4134)	2	3751	1776	1	0	0	45.77

Bioresponse	(4134)	2	3751	1776	1	0	0	45.77
Amazon_employee	(4135)	2	32769	0	10	0	0	5.79
PhishingWebsite	(4534)	2	11055	0	31	0	0	44.31
PhishingWebsite	(4534)	2	11055	0	31	0	0	44.31
GesturePhaseSeg	(4538)	5	9873	32	1	0	0	10.11
MiceProtein	(4550)	8	1080	77	5	1396	528	9.72
cylinder-bands	(6332)	2	540	18	22	999	263	42.22
cylinder-bands	(6332)	2	540	18	22	999	263	42.22
cjs	(23380)	6	2796	32	3	68100	2795	9.80
dresses-sales	(23381)	2	500	1	12	835	401	42.00
higgs	(23512)	2	98050	28	1	9	1	47.14
numerai28.6	(23517)	2	96320	21	1	0	0	49.48
LED-display-dom	(40496)	10	500	7	1	0	0	7.40
texture	(40499)	11	5500	40	1	0	0	9.09
Australian	(40509)	2	690	14	1	0	0	44.49
SpeedDating	(40536)	2	8378	59	64	18372	7330	16.47
connect-4	(40668)	3	67557	0	43	0	0	9.55
dna	(40670)	3	3186	0	181	0	0	24.01
shuttle	(40685)	7	58000	9	1	0	0	0.02
churn	(40701)	2	5000	16	5	0	0	14.14
Devnagari-Scrip	(40923)	46	92000	1024	1	0	0	2.17
CIFAR_10	(40927)	10	60000	3072	1	0	0	10.00
MiceProtein	(40966)	8	1080	77	5	1396	528	9.72
car	(40975)	4	1728	0	7	0	0	3.76
Internet-Advert	(40978)	2	3279	3	1556	0	0	14.00
mfeat-pixel	(40979)	10	2000	240	1	0	0	10.00
Australian	(40981)	2	690	6	9	0	0	44.49
steel-plates-fa	(40982)	7	1941	27	1	0	0	2.83
wilt	(40983)	2	4839	5	1	0	0	5.39
segment	(40984)	7	2310	19	1	0	0	14.29
climate-model-s	(40994)	2	540	20	1	0	0	8.52
Fashion-MNIST	(40996)	10	70000	784	1	0	0	10.00
jungle_chess_2p	(41027)	3	44819	6	1	0	0	9.67
APSFailure	(41138)	2	76000	170	1	1078695	75244	1.81
christine	(41142)	2	5418	1599	38	0	0	50.00
jasmine	(41143)	2	2984	8	137	0	0	50.00
sylvine	(41146)	2	5124	20	1	0	0	50.00
albert	(41147)	2	425240	26	53	2734000	425159	50.00
MiniBooNE	(41150)	2	130064	50	1	0	0	28.06
guillermo	(41159)	2	20000	4296	1	0	0	40.02
riccardo	(41161)	2	20000	4296	1	0	0	25.00
dilbert	(41163)	5	10000	2000	1	0	0	19.13
fabert	(41164)	7	8237	800	1	0	0	6.09
robert	(41165)	10	10000	7200	1	0	0	9.58
volkert	(41166)	10	58310	180	1	0	0	2.33
dionis	(41167)	355	416188	60	1	0	0	0.21
jannis	(41168)	4	83733	54	1	0	0	2.01
helena	(41169)	100	65196	27	1	0	0	0.17

Table 7: List of all tested data sets. Listed are the (abbreviated) name and OpenML id for each data set together with the number of classes, the number of samples, the number of numeric and categorical features per samples, how many values are missing in total (Missing values), how many samples contain at least one missing value (Incomp. Samples) and the percentage of samples belonging to the least frequent class (Minority %).

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 CASH benchmarking. Hyperparameter names equal the used names in scikit-learn. *cat* are categorical, *con* are continuous and *int* integer hyperparameters.

Appendix C. Raw Experiment Results

Data Set	Dummy	RF	Grid	Random	SMAC	BOHB	Optunity	hyperopt	RoBO	BTB
3	0.4991	0.9830	0.8488	0.9985	0.9983	0.9980	0.9979	**0.9989**	0.9975	0.9979
6	0.0396	0.9315	0.5482	0.9471	**0.9613**	0.9525	0.9459	0.9609	0.9438	0.9472
11	0.4394	0.8170	0.8718	0.9920	0.9867	0.9473	0.9660	**1.0000**	0.9862	0.9957
12	0.0997	0.9468	0.8542	0.9808	**0.9835**	0.9818	0.9800	0.9832	0.9833	0.9807
14	0.1065	0.7940	0.7498	0.8613	0.8560	0.8485	0.8625	**0.8678**	0.8635	0.8612
16	0.0982	0.8955	0.8442	0.9825	0.9815	0.9798	0.9793	**0.9827**	0.9813	0.9807
18	0.0988	0.7073	0.6788	0.7370	0.7443	0.7470	0.7378	**0.7478**	0.7303	0.7343
20	0.1023	0.9512	0.9212	0.9838	0.9843	0.9832	0.9823	**0.9855**	0.9823	0.9783
21	0.5414	0.9536	0.7582	0.9961	0.9940	0.9771	**0.9988**	0.9965	0.9882	0.9821
22	0.0995	0.7455	0.7050	0.8367	0.8360	0.8272	0.8345	0.8463	**0.8503**	0.8402
23	0.3597	0.5043	0.5063	0.5647	0.5622	0.5656	0.5636	**0.5853**	0.5695	0.5624
28	0.0992	0.9607	0.9057	0.9898	**0.9906**	0.9898	0.9897	0.9900	0.9901	0.9902
31	0.5837	0.7043	0.7053	0.7690	0.7697	0.7610	0.7743	**0.7753**	0.7617	0.7593
32	0.1006	0.9847	0.8008	0.9925	0.9938	0.9933	0.9924	**0.9939**	0.9936	0.9933
36	0.1414	0.9694	0.4338	0.9818	0.9818	0.9746	0.9838	**0.9857**	0.9788	0.9794
37	0.5403	0.7385	0.6489	0.7762	0.7883	0.7827	0.7823	**0.7996**	0.7861	0.7840
44	0.5206	0.9411	0.8888	0.9552	0.9542	0.9505	0.9566	**0.9581**	0.9503	0.9511
46	0.3814	0.9106	0.8361	0.9580	0.9580	0.9529	0.9619	**0.9654**	0.9479	0.9595
50	0.5354	0.9128	0.6451	**1.0000**	0.9983	0.9778	0.9972	**1.0000**	0.9962	0.9979
54	0.2492	0.7287	0.4307	0.8413	0.8406	0.8260	0.8362	0.8516	**0.8594**	0.8094
60	0.3369	0.8136	0.7111	0.8692	0.8709	0.8696	**0.8713**	0.8701	0.8697	0.8697
151	0.5106	0.8863	0.5935	0.9275	0.9183	0.9125	0.9302	**0.9377**	0.8852	0.9303
182	0.1923	0.8966	0.7091	0.9138	0.9171	0.9125	**0.9186**	0.9164	0.9073	0.9136
300	0.0370	0.8979	0.8432	0.9676	0.9683	0.9683	0.9654	**0.9718**	0.9578	0.9705
307	0.0882	0.9000	0.2633	0.9690	0.9822	0.9737	0.9731	0.9704	**0.9902**	0.9764
312	0.7105	0.8874	0.9303	0.9881	0.9881	0.9881	0.9876	**0.9906**	0.9893	0.9905
333	0.4934	0.9641	0.7413	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**
334	0.5464	0.8597	0.6497	0.9923	0.9818	0.9193	0.9917	**1.0000**	0.9934	0.9923
335	0.4976	0.9695	0.7431	0.9874	0.9868	0.9838	0.9868	**0.9898**	**0.9898**	0.9850
375	0.1144	0.9472	0.4545	0.9677	**0.9849**	0.9664	0.9733	0.9791	0.9686	0.9706
377	0.1689	0.9522	0.1706	0.9928	0.9944	0.9928	0.9922	0.9956	**0.9967**	0.9900
458	0.3229	0.9830	0.9783	0.9976	0.9988	0.9984	0.9984	**0.9992**	0.9988	0.9988
469	0.1692	0.1896	0.2325	0.2579	0.2612	0.2650	0.2621	**0.2692**	0.2596	0.2633
478	0.0893	0.7187	0.6093	0.9987	0.9920	0.9747	0.9867	**1.0000**	0.9953	0.9920
554	0.1010	0.9442	0.8331	0.9477	0.9445	0.9376	0.9357	**0.9578**	0.9403	0.9468

1036	0.8842	0.9871	0.9911	0.9950	0.9948	0.9944	**0.9952**	0.9948	0.9945	0.9941
1038	0.5014	0.9065	0.8012	0.9376	0.9375	0.9335	0.9423	**0.9516**	0.9302	0.9418
1043	0.6270	0.8297	0.7879	0.8521	0.8524	0.8500	0.8517	0.8565	0.8486	**0.8568**
1046	0.5582	0.9492	0.9353	0.9583	0.9580	0.9533	0.9583	**0.9605**	0.9538	0.9555
1049	0.7779	0.8975	0.8747	0.9178	0.9185	0.9153	0.9187	**0.9235**	0.9121	0.9151
1050	0.8158	0.8893	0.8663	0.9053	0.9068	0.9053	0.9053	**0.9100**	0.8983	0.9051
1063	0.6828	0.8127	0.8299	0.8669	**0.8707**	0.8650	0.8688	0.8669	0.8643	0.8586
1067	0.7409	0.8504	0.8509	0.8649	0.8660	0.8621	0.8640	0.8687	0.8657	**0.8727**
1068	0.8670	0.9330	0.9261	0.9396	0.9402	0.9363	0.9381	0.9432	**0.9438**	0.9372
1120	0.5455	0.8664	0.6491	0.8790	0.8797	0.8766	0.8802	**0.8819**	0.8714	0.8794
1169	0.5060	0.6144	0.5545	0.6650	0.6655	0.6635	0.6639	**0.6655**	0.6627	0.6627
1459	0.1017	0.8557	0.2446	0.8834	0.8631	0.8315	**0.9303**	0.9023	0.8623	0.8973
1461	0.7935	0.8991	0.8687	0.9079	0.9078	0.9070	**0.9084**	0.9071	0.9052	0.9044
1462	0.5056	0.9925	0.8451	**1.0000**	**1.0000**	**1.0000**	0.9995	**1.0000**	**1.0000**	0.9995
1464	0.6418	0.7329	0.7676	0.7978	0.7973	0.7951	0.7938	0.8009	**0.8076**	0.7991
1466	0.1530	0.9983	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**
1467	0.8438	0.9037	0.9111	0.9179	0.9198	0.9167	0.9173	**0.9284**	0.9204	0.9247
1468	0.1139	0.8985	0.9586	0.9571	**0.9630**	0.9614	0.9562	0.9599	0.9617	0.9537
1471	0.5074	0.8915	0.5519	0.9522	**0.9741**	0.9729	0.9541	0.9726	0.9414	0.9459
1475	0.2441	0.5822	0.3670	0.6082	0.6003	0.5969	0.6068	**0.6209**	0.6031	0.5984
1476	0.1773	0.9919	0.2300	0.9927	0.9931	0.9907	0.9920	**0.9948**	0.9933	0.9912
1478	0.1684	0.9650	0.8509	0.9893	0.9908	0.9896	0.9857	**0.9916**	0.9873	0.9885
1479	0.5074	0.5459	0.7857	0.9354	0.9558	**0.9566**	0.9321	0.9492	0.9511	0.9431
1480	0.5909	0.7034	0.7069	0.7354	0.7394	0.7383	0.7400	**0.7550**	0.7417	0.7469
1485	0.4991	0.6191	0.5922	0.8351	0.8340	0.8232	0.8171	**0.8484**	0.8194	0.8367
1486	0.5927	0.9640	0.8404	0.9662	0.9645	0.9655	0.9655	**0.9683**	0.9634	0.9646
1487	0.8837	0.9435	0.9351	0.9460	0.9468	0.9447	0.9466	0.9482	**0.9501**	0.9470
1489	0.5838	0.8873	0.7588	0.9004	0.9002	0.8946	0.8986	**0.9028**	0.8990	0.8949
1491	0.0100	0.6177	**0.8252**	0.8096	0.8144	0.7929	0.8117	0.8094	0.8100	0.8010
1492	0.0100	0.5135	0.1219	0.5994	**0.6146**	0.6137	0.5842	0.6012	0.6094	0.5773
1493	0.0104	0.6412	0.7217	0.8135	0.8025	0.7858	**0.8138**	**0.8138**	0.8037	0.8027
1494	0.5634	0.8492	0.7924	0.8814	**0.8893**	0.8795	0.8823	0.8849	0.8760	0.8804
1497	0.3356	0.9908	0.5913	0.9979	0.9971	0.9962	0.9977	**0.9983**	0.9966	0.9975
1501	0.1008	0.8690	0.8559	0.9475	0.9513	0.9433	0.9406	**0.9536**	0.9333	0.9416
1504	0.5528	0.9758	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**
1505	0.0550	0.9900	0.1339	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**
1510	0.5485	0.9474	0.8936	0.9713	0.9713	0.9719	**0.9749**	0.9719	0.9731	0.9737
1515	0.0599	0.7971	**0.9029**	0.8959	0.8971	0.8884	0.8837	0.8779	0.8913	0.8738
1570	0.8988	0.9814	0.9450	0.9857	0.9863	0.9853	0.9841	**0.9864**	0.9848	0.9851
1596	0.3771	**0.9388**	0.6375	0.8603	0.9303	0.9356	0.9344	0.8933	0.7836	0.8638
4134	0.5109	0.7586	0.6604	0.7967	0.8017	0.7956	0.7937	**0.8058**	0.7942	0.7969
4134	0.5023	0.7674	0.6660	0.7950	0.7955	0.7856	0.7948	**0.8139**	0.7901	0.8026
4135	0.8914	0.9441	0.9413	0.9480	0.9477	0.9458	0.9473	**0.9501**	0.9488	0.9475
4534	0.5062	0.9696	0.9097	0.9695	0.9701	0.9692	0.9712	**0.9724**	0.9658	0.9694
4534	0.5018	0.9688	0.9115	0.9708	0.9698	0.9682	0.9711	**0.9726**	0.9646	0.9699
4538	0.2374	0.5936	0.3597	0.6505	**0.6876**	0.6674	0.6405	0.6755	0.6349	0.6469
23517	0.4987	0.5031	0.5140	0.5220	0.5225	0.5230	0.5221	0.5215	0.5236	**0.5236**
40496	0.0947	0.7000	0.7533	0.7653	0.7687	0.7627	0.7693	0.7653	**0.7713**	0.7573
40499	0.0888	0.9622	0.2067	0.9981	0.9981	0.9977	0.9976	**0.9988**	0.9979	0.9981
40509	0.5145	0.8667	0.8831	0.8937	0.8932	0.8903	0.8932	**0.8947**	0.8903	0.8889
40668	0.5035	0.7868	0.6364	0.8012	0.8023	0.7968	0.7986	**0.8084**	0.8027	0.8019
40670	0.3855	0.9182	0.9449	0.9635	0.9621	0.9616	0.9655	**0.9656**	0.9552	**0.9656**
40685	0.6439	0.9997	0.8191	0.9995	0.9997	0.9995	0.9996	**0.9998**	0.9996	0.9994
40701	0.7529	0.9476	0.8601	0.9591	0.9603	0.9585	0.9592	**0.9618**	0.9531	0.9561
40923	0.0213	0.7779	0.5717	0.7187	0.7562	0.7308	0.6277	**0.7879**	0.6694	0.6610
40927	0.0994	0.3510	0.2956	0.3726	0.3680	**0.3974**	0.3285	0.3744	0.3282	0.3142
40975	0.5395	0.9563	0.7597	0.9881	0.9911	0.9723	0.9956	**0.9963**	0.9873	0.9913
40978	0.7520	0.9735	0.9685	0.9780	0.9778	0.9754	0.9771	**0.9792**	0.9738	0.9744
40979	0.0962	0.9522	0.9185	0.9822	0.9825	0.9810	0.9823	**0.9865**	0.9777	0.9785
40981	0.5150	0.8459	0.8657	0.8865	0.8845	0.8792	0.8816	**0.8942**	**0.8942**	0.8845
40982	0.2310	0.7448	0.4407	0.7861	0.8005	0.7913	0.7962	**0.8014**	0.7772	0.7878

40983	0.8981	0.9791	0.9451	0.9851	0.9864	0.9860	0.9853	**0.9874**	0.9842	0.9857
40984	0.1423	0.9222	0.4307	0.9335	0.9325	0.9261	0.9349	**0.9408**	0.9355	0.9394
40994	0.8469	0.9191	0.9185	0.9673	**0.9710**	0.9611	0.9630	0.9648	0.9617	0.9586
40996	0.1014	0.8571	0.7158	0.8526	0.8610	0.8543	0.8570	**0.8656**	0.8520	0.8487
41027	0.4247	0.7878	0.6166	0.8697	0.8610	0.8550	0.8698	**0.8759**	0.8473	0.8605
41142	0.4954	0.6806	0.6603	0.7299	0.7294	0.7256	0.7294	**0.7363**	0.7346	0.7315
41143	0.5030	0.7769	0.7510	0.8248	**0.8253**	0.8192	0.8229	0.8247	0.8160	0.8184
41146	0.5004	0.9300	0.5080	0.9516	0.9501	0.9464	0.9518	**0.9527**	0.9441	0.9445
41150	0.5962	0.9238	0.7733	0.9316	0.9300	0.9293	0.9288	**0.9332**	0.9285	0.9303
41159	0.5211	0.7765	0.5849	0.7237	0.7617	0.7443	0.7329	**0.7973**	0.7118	0.7585
41161	0.6243	0.9351	0.7037	0.9863	0.9868	0.9863	0.9855	**0.9884**	0.9868	0.9868
41163	0.2001	0.9171	0.6670	0.9384	0.9473	0.9270	0.9295	**0.9485**	0.9401	0.9406
41164	0.1620	0.6657	0.6544	0.6864	**0.6951**	0.6892	0.6896	0.6924	0.6909	0.6935
41165	0.0989	0.3104	0.3271	0.3897	0.3654	0.3745	**0.4055**	0.4055	0.3956	0.3940
41166	0.1481	0.6116	0.3813	0.6439	0.6451	0.6328	0.6306	**0.6508**	0.6321	0.6349
41167	0.0029	**0.8720**	0.4201	0.7447	0.8553	0.8399	0.8603	0.8543	0.7388	0.8089
41168	0.3593	0.6588	0.5277	0.6887	0.6890	0.6850	0.6880	**0.6913**	0.6848	0.6886
41169	0.0225	0.2917	0.1725	0.3242	**0.3330**	0.3248	0.3202	0.3320	0.3235	0.3222
Average	0.3902	0.8335	0.6964	0.8746	0.8782	0.8725	0.8748	**0.8821**	0.8711	0.8732

Table 9: Average accuracy of CASH solvers on selected OpenML data sets. Data sets containing missing values are omitted. The best results per data set are highlighted in bold.

Data Set	Dummy	RF	Random	auto-sklearn	TPOT	ATM	hpsklearn	H2O
3	0.50761	0.98467	0.99062	0.98986	**0.99431**	0.99326	0.99051	0.99426
12	0.10317	0.94617	0.97633	0.97767	0.97333	**0.98178**	0.94758	0.97433
15	0.52857	0.95714	0.95873	0.96875	0.96571	**0.98474**	0.96000	0.96286
23	0.35249	0.50950	0.53262	0.54638	0.55882	**0.58100**	0.53047	0.53733
24	0.49922	**1.00000**	0.99993	**1.00000**	**1.00000**	**1.00000**	**1.00000**	0.99848
29	0.51111	0.84976	0.85507	0.87289	0.86377	**0.89133**	0.85956	0.86184
31	0.56867	0.72667	0.72400	0.73433	0.74400	**0.76578**	0.70121	0.74867
38	0.88207	0.98454	0.98550	0.98288	**0.98746**	–	0.97438	0.98419
42	0.08439	0.91561	0.91911	0.91954	0.92732	**0.94504**	0.92585	0.93122
54	0.26417	0.72165	0.81969	0.82008	0.81811	0.81522	0.75787	**0.82717**
188	0.21267	0.61086	0.62670	0.63886	0.65566	0.64190	0.64072	**0.65570**
451	0.50533	0.99933	0.99081	0.99019	0.99091	**1.00000**	0.99404	0.97967
469	0.16583	0.18625	0.20382	0.20365	0.20833	**0.27028**	0.19139	0.19542
470	0.56733	0.65050	0.64563	0.65687	0.66832	**0.71221**	0.63762	0.71089
1053	0.68766	0.80505	0.81126	0.81344	0.81810	**0.82100**	0.80998	0.74819
1067	0.74060	0.84739	0.85340	0.85118	0.86019	**0.86856**	0.84044	0.80869
1111	0.96487	0.98235	0.98228	**0.98244**	0.98182	–	0.98189	0.96555
1112	0.86358	0.92542	0.92586	**0.92725**	0.92624	–	0.92599	0.78802
1114	0.86357	0.94048	0.95030	**0.95094**	0.95085	–	0.95068	0.93415
1169	0.50570	0.61520	0.59845	0.66665	**0.66895**	0.63671	0.65080	0.61266
1461	0.79323	0.89985	0.90398	0.90447	**0.90705**	0.89957	0.90451	0.90060
1464	0.63200	0.74889	0.77778	0.76667	0.78711	**0.81956**	0.78044	0.73378
1468	0.10741	0.88765	0.93117	0.94167	0.94784	**0.96049**	0.94012	0.95216
1475	0.24553	0.58998	0.58601	0.59695	0.61291	0.60272	0.58293	**0.61656**
1486	0.59173	0.96344	0.96656	0.96903	0.97026	0.96055	0.96891	**0.97146**
1489	0.58453	0.88890	0.89205	0.89716	**0.90450**	0.89963	0.89273	0.89205
1492	0.00687	0.51333	0.62795	**0.65172**	0.61146	0.61097	0.54667	0.56435
1590	0.63379	0.85021	0.87013	0.86938	**0.87089**	0.85448	0.86727	0.86656
1596	0.37644	0.93818	0.89143	**0.96395**	0.94542	0.66390	0.95227	0.92908
4134	0.50462	0.76314	0.77762	0.78890	**0.80249**	0.77087	0.77798	0.80044
4135	0.88895	0.94491	0.94444	0.94761	0.94891	0.94606	0.94750	**0.95114**
4534	0.50612	0.96847	0.96244	0.96590	0.96913	0.96464	0.96964	**0.97160**
4538	0.23130	0.59207	0.65004	0.67733	0.67586	0.66217	0.67272	**0.70165**
4550	0.12346	0.99414	0.99907	**1.00000**	**1.00000**	**1.00000**	0.99983	**1.00000**

6332	0.52407	0.73951	0.76173	0.79012	0.81009	**0.81701**	0.76667	0.78333
6332	0.49877	0.76481	0.77058	0.77353	**0.81173**	0.79155	0.75823	0.80000
23380	0.18677	0.95000	0.99841	0.98265	**1.00000**	–	0.97131	**1.00000**
23381	0.50333	0.55867	0.55556	0.56667	0.56867	**0.66978**	0.56844	0.58400
23512	0.50065	0.67445	0.71930	**0.72296**	0.72031	0.67135	0.70743	0.71281
23517	0.49962	0.50259	0.51939	0.51926	**0.52082**	0.51941	0.52033	0.50635
40536	0.72550	0.85195	0.86225	0.86291	0.86392	0.86128	**0.86661**	0.84968
40668	0.50439	0.78341	0.79628	0.82109	0.84123	0.77698	0.82886	**0.86500**
40670	0.39100	0.91412	0.95889	0.95962	0.95931	0.95282	0.96109	**0.96904**
40685	0.64405	0.99962	0.99968	0.99978	0.99974	0.99955	0.99253	**0.99987**
40701	0.76320	0.94313	0.95313	0.95620	**0.96000**	0.95007	0.94533	0.95370
40923	0.02127	0.78048	0.02169	0.74009	–	**0.89470**	0.86438	0.58220
40927	0.10096	0.35102	–	–	0.29429	0.32001	0.32093	**0.36389**
40966	0.12407	0.94228	0.99506	0.99043	0.99506	**1.00000**	0.96380	0.99551
40975	0.53218	0.95318	0.97958	0.97264	**0.99422**	0.96763	0.98786	0.99191
40978	0.75346	0.97368	0.97114	**0.97774**	0.97398	0.96900	0.97358	–
40979	0.09983	0.95217	0.97367	0.97783	0.96883	0.97750	**0.98121**	0.97600
40981	0.49324	0.85604	0.85556	0.87053	0.86184	**0.89050**	0.86913	0.87633
40982	0.21681	0.74425	0.76364	0.78268	**0.79091**	0.76415	0.75955	0.78062
40983	0.89683	0.97886	0.98581	0.98612	0.98540	**0.98657**	0.95289	0.98574
40984	0.14473	0.93001	0.93333	0.93088	0.94055	0.92564	0.90664	**0.94185**
40994	0.83704	0.91914	0.92407	0.94074	0.94547	**0.96975**	0.92593	0.93642
40996	0.09844	0.85777	0.84450	**0.87844**	0.78089	0.82114	0.85060	0.87341
41027	0.42598	0.78945	0.85378	0.86775	0.88735	0.87540	0.88691	**0.90047**
41138	0.96474	0.99268	0.99137	0.99287	0.99339	0.97097	0.99360	**0.99369**
41142	0.50234	0.67977	0.73081	**0.74754**	0.72645	0.72169	0.71630	0.72811
41143	0.50748	0.78170	0.80603	0.82009	**0.82366**	0.79911	0.80078	0.80906
41146	0.49532	0.93062	0.94753	0.93921	**0.95533**	0.93476	0.94675	0.92510
41147	0.49923	0.62564	0.66709	0.68314	0.66110	**0.80064**	0.66694	0.64798
41150	0.59589	0.92356	0.92891	0.94334	0.93850	0.90234	0.87477	**0.94604**
41159	0.51942	0.77610	–	0.64227	0.72548	0.66063	0.74347	**0.81928**
41161	0.62482	0.93468	0.75042	0.74757	**0.98495**	0.90729	0.82518	0.95625
41163	0.19703	0.92263	0.94793	**0.98357**	0.96254	0.95391	0.97243	0.96988
41164	0.16375	0.66570	0.67395	0.70255	0.68336	0.67357	0.69104	**0.71752**
41165	0.09480	0.30877	0.39922	**0.44843**	–	0.35252	0.34203	–
41166	0.14885	0.61045	0.63762	0.66933	0.65075	**0.67940**	0.65451	0.67841
41167	0.00286	**0.87164**	–	–	–	0.38666	0.77971	–
41168	0.36200	0.65848	0.69273	**0.71814**	0.69642	0.63788	0.68494	0.71786
41169	0.02272	0.29082	0.29566	0.30692	**0.33576**	0.32108	0.28741	–
Average	0.44921	0.79980	0.80853	0.82606	**0.83040**	0.80292	0.81075	0.82910

Table 10: Average accuracy of AutoML frameworks on selected OpenML data sets. Entries marked by – consistently failed to generate an ML pipeline. The best results per data set are highlighted in bold.

References

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.

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/`.

Adithya Balaji and Alexander Allen. Benchmarking Automatic Machine Learning Frameworks. *arXiv preprint arXiv:1808.06492*, 2018.

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

Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, James Luedtke, and Ashutosh Mahajan. Mixed-integer nonlinear optimization. *Acta Numerica*, 22:1–131, 2013.

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.

Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. OpenML Benchmarking Suites. *arXiv preprint arXiv:1708.03731v2*, 2019.

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

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.

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.

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

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. *International Joint Conference on Artificial Intelligence*, pages 3460–3468, 2015. ISSN 10450823.

Ofer Dor and Yoram Reich. Strengthening learning algorithms by feature discovery. *Information Sciences*, 189:176–190, 2012.

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.

Simão Eduardo and Charles Sutton. Data Cleaning using Probabilistic Models of Integrity Constraints. In *Neural Information Processing Systems*, 2016.

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.

Radwa Elshawi, Mohamed Maher, and Sherif Sakr. Automated Machine Learning: State-of-The-Art and Open Challenges. *arXiv preprint arXiv:1906.02287*, 2019.

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.

M. Giselle Fernández-Godino, Chanyoung Park, Nam-Ho Kim, and Raphael T. Haftka. Review of multi-fidelity models. *arXiv preprint arXiv:1609.07196*, 2016.

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.

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

Keinosuke Fukunaga and Larry D. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. AJAX:An Extensible Data Cleaning Tool. In *International Conference on Management of Data*, pages 590–596, 2000.

Joao Gama and Pavel Brazdil. Characterization of Classification Algorithms. In *Portuguese Conference on Artificial Intelligence*, 2000.

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.

Romaric Gaudel and Michèle Sebag. Feature Selection as a One-Player Game. In *International Conference on Machine Learning*, pages 359–366, 2010.

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.

Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An Open Source AutoML Benchmark. In *International Conference on Machine Learning AutoML Workshop*, 2019.

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.

Taciana A.F. Gomes, Ricardo B.C. Prudêncio, Carlos Soares, André L.D. Rossi, and André Carvalho. Combining Meta-Learning and Search Techniques to Select Parameters for Support Vector Machines. *Neurocomputing*, 75(1):3–13, 2012.

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

John Clifford Gower. A General Coefficient of Similarity and Some of Its Properties. *Biometrics*, 27(4):857–871, 1971.

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/`.

H2O.ai. H2O AutoML, 2019. URL `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`.

Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A Survey of the State-of-the-Art. *arXiv preprint arXiv:1908.00709*, 2019.

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. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018a.

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

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.

Zuhair Khayyaty, Ihab F. Ilyasz, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. BigDansing: A System for Big Data Cleansing. In *ACM International Conference on Management of Data*, pages 1215–1230, 2015.

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

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning Curve Prediction With Bayesian Neural Networks. *International Conference on Learning Representations*, pages 1–16, 2017b.

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.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

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.

Igor Kononenko. Estimating attributes: Analysis and extensions of RELIEF. *European Conference on Machine Learning*, 1994.

Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *Journal of Machine Learning Research*, 17:1–5, 2016.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. ISBN 0-262-11170-5.

Sanjay Krishnan and Eugene Wu. AlphaClean: Automatic Generation of Data Cleaning Pipelines. *arXiv preprint arXiv:1603.06560*, 2019.

Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, Ken Goldberg, Tim Kraska, Tova Milo, and Eugene Wu. SampleClean: Fast and Reliable Analytics on Dirty Data. *IEEE Data Engineering Bulletin*, 38(3):59–75, 2015.

Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. ActiveClean: Interactive Data Cleaning For Statistical Modeling. In *Proceedings of the VLDB Endowment*, volume 12, pages 948–959, 2016.

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.

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.

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.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

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.

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.

Dimitris Margaritis. Toward Provably Correct Feature Selection in Arbitrary Domains. In *Neural Information Processing Systems*, pages 1240–1248, 2009. ISBN 9781615679119.

Shaul Markovitch and Dan Rosenstein. Feature generation using general constructor functions. *Machine Learning*, 49(1):59–98, 2002.

Oded Maron and Aw Moore. Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. *Advances in Neural Information Processing Systems*, pages 59–66, 1993.

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

Nicolai Meinshausen and Peter Bühlmann. Stability selection. *Journal of the Royal Statistical Society*, 72(4):417–473, 2010.

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.

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

Hiroshi Motoda and Huan Liu. Feature Selection, Extraction and Construction. *Communication of Institute of Information and Computing Machinery*, 5:67–72, 2002.

Rémi Munos. From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning. Technical report, hal-00747575, 2014.

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`.

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.

Yao Quanming, Wang Mengshuo, Jair Escalante Hugo, Guyon Isabelle, Hu Yi-Qi, Li Yu-Feng, Tu Wei-Wei, Yang Qiang, and Yu Yang. Taking Human out of Learning Applications: A Survey on Automated Machine Learning. *arXiv preprint arXiv:1810.13306*, 2018.

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

Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. Automated Machine Learning with Monte-Carlo Tree Search. In *International Joint Conference on Artificial Intelligence*, pages 3296–3303, 2019.

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.

John W. Ratcliff and David E. Metzener. Pattern Matching: The Gestalt Approach. *Dr Dobbs Journal*, 13(7):46–72, 1988.

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

Theodoros Rekatsinas, Xu Chuy, Ihab F. Ilyasy, and Christopher Ré. HoloClean: Holistic Data Repairs with Probabilistic Inference. In *VLDB Endowment*, pages 1190–1201, 2017.

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.

Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

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.

Colin Shearer. The CRISP-DM model: the new blueprint for data mining. *Journal of Data Warehousing*, 5(4):13–22, 2000.

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.

Matthew G. Smith and Larry Bull. Genetic Programming with a Genetic Algorithm for Feature Construction and Selection. *Genetic Programming and Evolvable Machines*, 6 (3):265–281, 2005. ISSN 13892576.

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.

So Young Sohn. Meta analysis of classification algorithms for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1137–1144, 1999. ISSN 01628828. doi: 10.1109/34.809107.

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

Parikshit Sondhi. Feature Construction Methods: A Survey. *Sifaka. Cs. Uiuc. Edu*, 69: 70–71, 2009.

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.

Binh Tran, Bing Xue, and Mengjie Zhang. Genetic programming for feature construction and selection in classification on high-dimensional data. *Memetic Computing*, 8:3–15, 2016.

Lukas Tuggener, Mohammadreza Amirian, Katharina Rombach, Stefan Lörwald, Anastasia Varlet, Christian Westermann, and Thilo Stadelmann. Automated Machine Learning in Practice: State of the Art and Recent Results. *arXiv preprint arXiv:1907.08362*, 2019.

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

Haleh Vafaie and Kenneth De Jong. Genetic Algorithms as a Tool for Feature Selection in Machine Learning. In *International Conference on Tools with Artificial Intelligence*, pages 200–203, 1992.

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: A Survey. *CoRR*, abs/1810.0:1–29, 2018.

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.

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.

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

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/`.