Benchmark and Survey of Automated Machine Learning Frameworks

Marc-André Zöller   MARC.ZOELLER@USU.COM
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, Allmandring 25, Stuttgart, Germany &
Fraunhofer Institute for Manufacturing Engineering and Automation IPA
Nobelstr. 12, Stuttgart, Germany

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 build machine learning applications automatically 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 data sets from established AutoML benchmark suites.

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. Such 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 not a new trend. Starting from the 1990s, commercial solutions offered automatic HPO for selected classification algorithms via grid search (Dinsmore, 2016). Adaptations of grid search to test possible configurations in a greedy best-first approach are available since 1995 (Kohavi & John, 1995). In the early 2000s, the first efficient strategies for HPO have been proposed. For limited settings, e.g., tuning C and γ of a support-vector machine (SVM) (Momma & Bennett, 2002; Chapelle et al., 2002; 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 build a complete ML pipeline automatically by selecting a preprocessing, feature selection and classification algorithm simultaneously 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 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 & Veeramachaneni, 2015). Building variable shaped pipelines is possible since 2016 (Olson & Moore, 2016). In 2017 and 2018, the topic AutoML received a lot of attention in the media with the release of commercial AutoML solutions from various global players (Golovin et al., 2017; Clouder, 2018; Baidu, 2018; Das et al., 2020). 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 frameworks for AutoML and HPO on real data. We select 14 different AutoML and HPO frameworks in total for evaluation. The 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 structure, selecting an ML algorithm for each stage in a pipeline and tuning each algorithm. The paper focuses on classic 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 are:

- We introduce a mathematical formulation covering the complete procedure of automatic ML pipeline synthesis and compare it with existing problem formulations.

- We review open-source frameworks for building ML pipelines automatically.

- An evaluation of eight HPO algorithms on 137 real data sets is conducted. To the best of our knowledge, this is the first independent benchmark of HPO algorithms.

- An empirical evaluation of six AutoML frameworks on 73 real data sets is performed. To the best of our knowledge, this is the most extensive evaluation—in terms of tested frameworks 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 raising 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.

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. Opportunities for further research are presented in Section 10 followed by a short conclusion in Section 11.

2. Problem Formulation

An ML pipeline $h : \mathbb{X} \to \mathbb{Y}$ is a sequential combination of various algorithms that transforms a feature vector $\vec{x} \in \mathbb{X}$ 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 $\Lambda_{A^{(i)}}$.

Without loss of generality, let a pipeline structure be modeled as a 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 structure 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 $g \in G$.

Definition 1 (Machine Learning Pipeline) *Let a triplet $(g, \vec{A}, \vec{\lambda})$ define an ML pipeline with $g \in G$ a valid pipeline structure, $\vec{A} \in \mathcal{A}^{|g|}$ 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 $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$.*

Following the notation from empirical risk minimization, let $P(\mathbb{X}, \mathbb{Y})$ be a joint probability distribution of the feature space $\mathbb{X}$ and target space $\mathbb{Y}$ known as a *generative model*. We denote a pipeline trained on the generative model P as $\mathcal{P}_{g,\vec{A},\vec{\lambda},P}$.

Definition 2 (True Pipeline Performance) *Let a pipeline $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$ be given. Given a loss function $\mathcal{L}(\cdot, \cdot)$ and a generative model $P(\mathbb{X}, \mathbb{Y})$, the performance of $\mathcal{P}_{g,\vec{A},\vec{\lambda},P}$ is calculated as*

$$R\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},P}, P\right) = \mathbb{E}\big(\mathcal{L}(h(\mathbb{X}), \mathbb{Y})\big) = \int \mathcal{L}\big(h(\mathbb{X}), \mathbb{Y}\big)\, \mathrm{d}P(\mathbb{X}, \mathbb{Y}), \tag{1}$$

with $h(\mathbb{X})$ being the predicted output of $\mathcal{P}_{g,\vec{A},\vec{\lambda},P}$.

Let an *ML task* be defined by a generative model, loss function and an ML problem type, e.g., classification or regression. Generating an ML pipeline for a given ML task can be split into three tasks: first, the structure of the pipeline has to be determined, e.g., selecting how many 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 step an algorithm has to be selected. Finally, for each selected algorithm its corresponding hyperparameters have to be selected. All 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 $\Lambda_{(\cdot)}$, a set of valid pipeline structures G and a generative model $P(\mathbb{X}, \mathbb{Y})$ be given. The pipeline creation problem consists of finding a pipeline structure in combination with a joint algorithm and hyperparameter selection that minimizes the loss*

$$(g, \vec{A}, \vec{\lambda})^\star \in \underset{g \in G, \vec{A} \in \mathcal{A}^{|g|}, \vec{\lambda} \in \Lambda}{\arg\min} R\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},P}, P\right). \tag{2}$$

In general, Equation (2) cannot be computed directly as the distribution $P(\mathbb{X}, \mathbb{Y})$ is unknown. Instead, let a finite set of observations $D = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$ of m i.i.d samples drawn from $P(\mathbb{X}, \mathbb{Y})$ be given. Equation (1) can be adapted to D to calculate an *empirical pipeline performance* as

$$\hat{R}\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},D}, D\right) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(h(x_i), y_i\right). \tag{3}$$

To limit the effects of overfitting, Equation (3) is often augmented by cross-validation. Let the data set D be split into k folds $\{D_{\text{valid}}^{(1)}, \ldots, D_{\text{valid}}^{(k)}\}$ and $\{D_{\text{train}}^{(1)}, \ldots, D_{\text{train}}^{(k)}\}$ such that $D_{\text{train}}^{(i)} = D \setminus D_{\text{valid}}^{(i)}$. The final objective function is defined as

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

This problem formulation is a generalization of existing problem formulations. Current problem formulations only consider selecting and tuning a single algorithm (e.g., Escalante et al., 2009; Bergstra et al., 2011) or a linear sequence of algorithms with (arbitrary but) fixed length (e.g., Thornton et al., 2013; Zhang et al., 2016; Alaa & Van Der Schaar, 2018; Hutter et al., 2018a). Salvador et al. (2017) model an ML pipeline with Petri-nets (Petri, 1962) instead of a DAG. Using additional constraints, the Petri-net is enforced to represent a DAG. Even though this approach is more expressive than DAGs, the additional model capabilities are currently not utilized in the context of AutoML.

Using Equation (2), 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 solve specific problem instances efficiently, 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 generative model. Consequently, efficient solvers, like convex or gradient-based optimization, cannot be used for Equation (2) (Luo, 2016).

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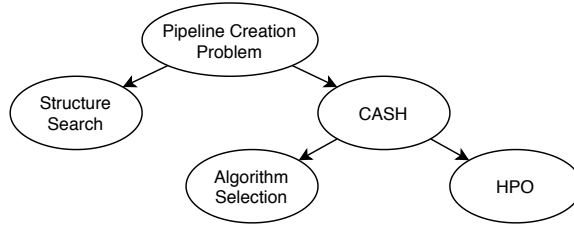

Figure 1: Subproblems of the pipeline creation problem.

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. In practice this simple pipeline is usually adapted and extended by experienced data scientists.
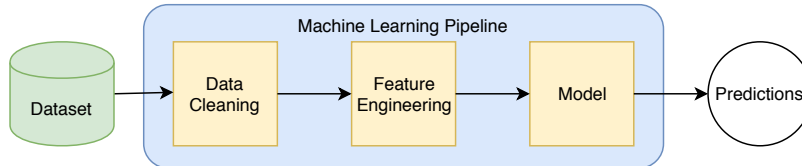

Figure 2: Prototypical ML pipeline. First, the input data is cleaned and features are extracted. The transformed input is passed through an ML model to create predictions.

3.1 Fixed Structure

Many AutoML frameworks do not solve the structure selection because they are preset to the fixed pipeline structure displayed in Figure 3 (e.g., Komer et al., 2014; Feurer et al., 2015a; Swearingen et al., 2017; Parry, 2019; McGushion, 2019). Resembling the best practice pipeline closely, the pipeline is a linear sequence of multiple data cleaning steps, a feature selection step, one variable preprocessing step and exactly one modeling step. The preprocessing step chooses one algorithm from a set of well known algorithms, e.g., various matrix decomposition algorithms. Regarding data cleaning, the pipeline structure differs. Yet, often the two steps imputation and scaling are implemented. Often single steps in this pipeline could be omitted as the data set is not affected by this specific step, e.g., an imputation without missing values.
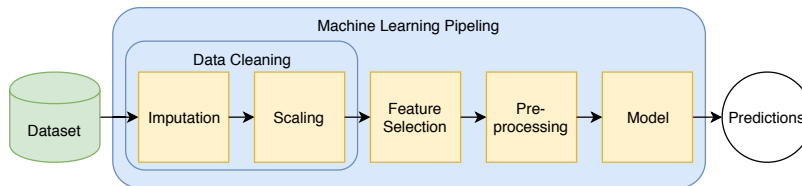

Figure 3: Fixed ML pipeline used by most AutoML frameworks. Minor differences exist regarding the implemented data cleaning steps.

By using a pipeline with a fixed structure, the complexity of determining a graph structure g is eliminated completely 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 may lead to inferior pipeline performances for complex data sets requiring, for example, multiple preprocessing steps. Yet, for many problems with high quality training data a simple pipeline structure may still be sufficient.

3.2 Variable Structure

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 building flexible pipelines automatically 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 *data set duplicator* to clone a data set and a *feature union* operator to combine multiple data sets. 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) and is based on genetic programming (Koza, 1992; Banzhaf, Nordin, Keller, & Francone, 1997). Genetic programming has been used for automatic program code generation for a long time (Poli et al., 2008). Yet, the application to pipeline structure synthesis 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 is implemented by random addition or deletion of a node. This way, flexible pipelines can be generated.
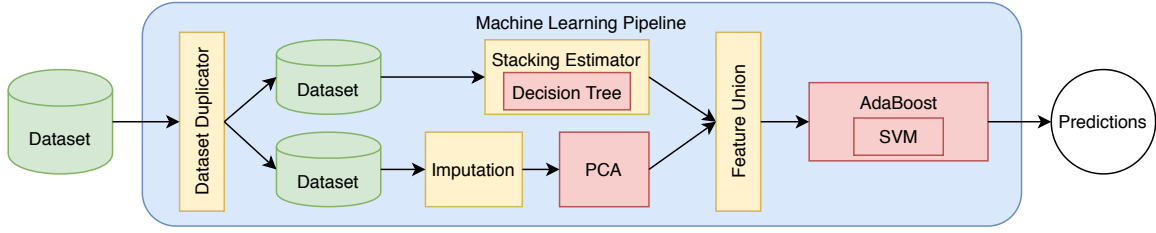
Figure 4: Specialized ML pipeline for a specific ML task.

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 & Szepesvári, 2006; Browne et al., 2012) is a heuristic best-first tree search algorithm. Similar to hierarchical planning, ML pipeline structure generation is reduced to finding the best node in the search tree. However, instead of decomposing complex tasks, pipelines with increasing complexity are created iteratively (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; for each step the player can choose between the three actions adding, removing or replacing a single node 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 structure g by predicting its performance and probabilities which action to chose in this state (Drori et al., 2018).

Methods for variable-shaped pipeline construction often do not consider dependencies between different pipeline stages and constraints on the complete pipeline. For example, genetic programming could create a pipeline for a classification task without any classification algorithm (Olson et al., 2016). To prevent such defective pipelines, the pipeline creation can be restricted by a grammar (de Sá et al., 2017; Drori et al., 2019). In doing so, reasonable but still flexible pipelines can be created.

4. Algorithm Selection and Hyperparameter Optimization

Let a structure $g \in G$, 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 (2). For readability, assume $|g| = 1$. The CASH problem is defined as

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

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

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

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

$$\vec{\lambda}^\star \in \underset{\vec{\lambda} \in \Lambda}{\arg\min} \ R\left(\mathcal{P}_{g,\vec{\lambda},D}, D\right). \tag{4}$$

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

It is important to note that Equation (4) is not easily solvable as the search space is quite large and complex. As hyperparameters can be categorical and real-valued, Equation (4) 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 $\Lambda_{A^{(i)}}$ is relevant as all other hyperparameters do not influence the result. Therefore, $\Lambda_{A^{(i)}}$ depends on $\lambda_r = i$. Following Hutter et al. (2009), Thornton et al. (2013), Swearingen et al. (2017) the hyperparameters $\vec{\lambda} \in \Lambda_{A^{(i)}}$ 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 another 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 (4).

4.1 Grid Search

The first approach to explore the configuration space systematically was grid search. As the name implies, grid search creates a grid of configurations and evaluates all of them. Even though grid search is easy to implement and parallelize (Bergstra & 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
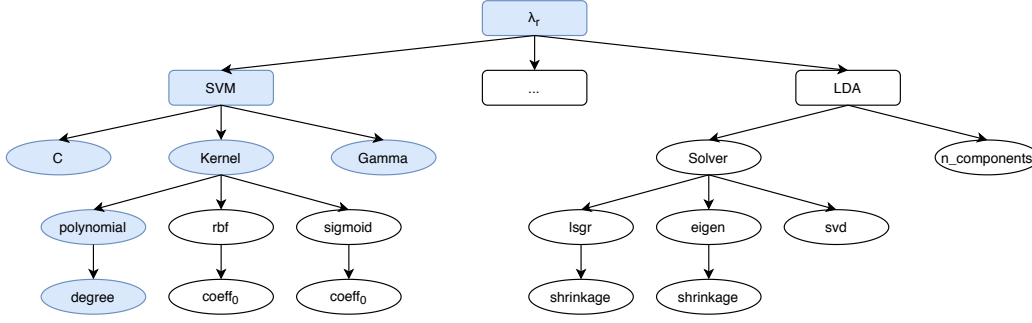
Figure 5: Incomplete representation of the structured configuration space for selecting and tuning a classification algorithm. Rectangle nodes represent the selection of an algorithm. Ellipse nodes represent tunable hyperparameters. Highlighted in blue is an active configuration to select and configure a SVM with a polynomial kernel.

et al., 2004) and 2) the hierarchical hyperparameter structure is not considered, leading to many redundant configurations.

In the traditional version, grid search does not exploit knowledge of well performing regions. This drawback is partially eliminated by *contracting* 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 choosing a value for each hyperparameter randomly and independently of all others. Conditional hyperparameters can be handled implicitly 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 & Wets, 1981). Even though the convergence speed is faster than grid search (Bergstra & 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: $f(\vec{\lambda})$ can be approximated using standard regression methods based on the so-far tested hyperparameter configurations $D_{1:n} = \left\{ \left(\vec{\lambda}_1, f(\vec{\lambda}_1) \right), \ldots, \left(\vec{\lambda}_n, f(\vec{\lambda}_n) \right) \right\}$. 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.

The loss function is complemented by a probabilistic regression model M that acts as a surrogate for f. The surrogate model M, build using $D_{1:n}$, allows predicting the performance of an arbitrary configuration $\vec{\lambda}$ without evaluating the demanding objective function. A new
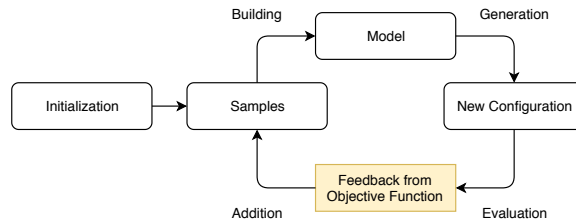
Figure 6: Schematic procedure of SMBO.

configuration $\vec{\lambda}_{n+1} \in \Lambda$, obtained using a cheap acquisition function, 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 badly performing configurations can be lowered significantly. As the actual function evaluation is usually way more expensive than 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 (Brochu et al., 2010; Shahriari et al., 2016; Frazier, 2018) is used. It is an iterative optimization framework being well suited for expensive objective functions. A probabilistic model of the objective function f is obtained using Bayes' theorem

$$P\left(f \mid D_{1:n}\right) \propto P\left(D_{1:n} \mid f\right) P(f). \tag{5}$$

Bayesian optimization is very efficient concerning the number of objective function evaluations (Brochu et al., 2010) as the acquisition function handles the trade-off between exploration and exploitation automatically. New regions with a high uncertainty are explored, preventing the optimization from 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 (5). As the characteristics and shape of the loss function are in general unknown, the posterior has to be a non-parametric model.

The traditional surrogate models for Bayesian optimization are Gaussian processes (Rasmussen & 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 & 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

---

1. Extensions for treating integer variables in Gaussian processes exist (e.g., Levesque et al., 2017; Garrido-Merchán & Hernández-Lobato, 2018).

data to create groups of similar observations. Besides the ability to handle categorical variables natively, random forests are fast to train and even faster on evaluating new data while obtaining a good predictive power.

In contrast to the two previous surrogate models, a tree-structured Parzen estimator (TPE) (Bergstra et al., 2011) models the likelihood $P(D_{1:n} \mid f)$ instead of the posterior. Using a performance threshold f', all observed configurations are split into a well and badly performing set, respectively. Using kernel density estimation (KDE) (Parzen, 1961), those sets are transformed into two distributions. Regarding the tree structure, TPEs handle hierarchical search spaces natively 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 & 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 hierarchical 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 instances—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 to improve 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 & Do, 2000). In general, data cleaning is split into two tasks: error detection and error repairing (Chu et al., 2016). For over two decades semi-automatic, interactive systems existed to aid a data scientist in data cleaning (Galhardas et al., 2000; Raman & Hellerstein, 2001). Yet, most current approaches still aim to assist a human data scientist instead of fully automated data cleaning, (e.g., Krishnan et al., 2015; Khayyaty et al., 2015; Krishnan et al., 2016; Eduardo & Sutton, 2016; Rekatsinas et al., 2017). Krishnan and Wu (2019) proposed an automatic data cleaning procedure with minimal human interaction: based on a human defined *data quality* function, data cleaning is treated similarly to pipeline structure search. Basic data cleaning operators are combined iteratively 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 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 methods.

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; Nguyen et al., 2020).

Incorporating domain knowledge during data cleaning increases the data quality significantly (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 detected and corrected automatically (Raman & 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. 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 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 an identical pipeline without dedicated feature engineering (Pyle, 1999). Feature engineering can be split into three sub-tasks: feature extraction, feature construction and feature selection (Motoda & Liu, 2002). Feature engineering—especially feature construction—is highly domain specific and difficult to generalize. Even for data scientists 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 & 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 & 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 transformed easily without domain knowledge to extract more meaningful features (Chen et al., 2018).
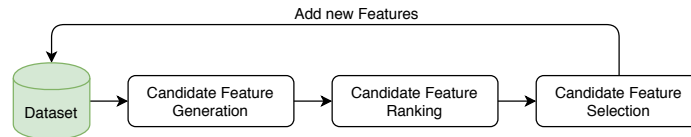
Figure 7: Iterative feature generation procedure.

Basically all automatic feature generation approaches follow the iterative schema displayed in Figure 7. Based on an initial data set, a set of candidate features is generated and ranked. Highly ranked features are evaluated and added to the data set potentially. 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 &

Rosenstein, 2002). More recently, greedy search (Dor & 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 & Bull, 2005; Tran et al., 2016).

Instead of exploring the transformation tree iteratively, exhaustive approaches consider a fully expanded transformation tree up to a predefined depth (Kanter & 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., 2018b) 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 & Veeramachaneni, 2015; Lam et al., 2017). Wistuba et al. (2017) and Chen et al. (2018) propose 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. (2018a) proposed to create an ensemble of sub-optimal feature sets (see Section 7.4).

Another approach for automatic feature generation is *representation learning* (Bengio et al., 2013; Goodfellow et al., 2016). Representation learning aims to transform the input data into a latent representation space well suited for a—in the context of this survey—supervised learning task automatically. As this approach is usually used in combination with neural networks and unstructured data, it is not further evaluated.

6.2 Feature Selection

Feature selection chooses a subset of the feature set to speed up the subsequent ML model training and to improve its performance by removing redundant or misleading features (Motoda & Liu, 2002). Furthermore, the interpretability of the trained model is increased. Simple domain-agnostic filtering approaches for feature selection are based on information theory and statistics (Pudil et al., 1994; Yang & Pedersen, 1997; Dash & Liu, 1997; Guyon & Elisseeff, 2003). Algorithms like univariate selection, variance threshold, feature importance, correlation matrices (Saeys et al., 2007) or stability selection (Meinshausen & Bühlmann, 2010) are already integrated in modern AutoML frameworks (Thornton et al., 2013; Komer et al., 2014; Feurer et al., 2015a; Olson & 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 & 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, *embedded* methods incorporate feature selection directly into the training process of an ML model. Many ML models provide some sort of feature ranking that can be utilized, e.g., SVMs (Guyon et al., 2002; Rakotomamonjy, 2003), perceptrons (Mejía-Lavalle et al., 2006) or random forests (Tuv et al., 2009). Similarly, embedded methods can be used 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 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, badly performing configurations can be discarded quickly 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 (2) by introducing an additional budget term $s \in [0, 1]$ that can be freely selected by the optimization algorithm.

SUCCESSIVEHALVING (Jamieson & Talwalkar, 2015) solves the selection of s via bandit learning. The basic idea, as visualized in Figure 8, is 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
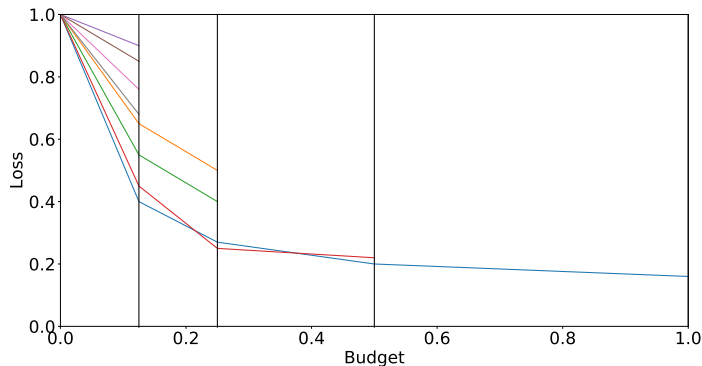
Figure 8: Schematic representation of SUCCESSIVEHALVING with 8 different configurations.

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 selecting an appropriate number of configurations dynamically. It calculates the number of configurations and budget size based on some budget constraints. A descending sequence of configuration numbers m is calculated and passed to SUCCESSIVEHALVING. Consequently, no prior knowledge is required anymore for SUCCESSIVEHALVING.

FABOLAS (Klein et al., 2016) treats the budget s as an additional input parameter in the search space that can be freely chosen by the optimization procedure instead of being deterministically calculated. A Gaussian process is trained on the combined input $(\vec{\lambda}, s)$. Additionally, the acquisition function is enhanced by entropy search (Hennig & Schuler, 2012). This allows predicting the performance of $\vec{\lambda}_i$, tested with budget s_i, for the full budget $s = 1$.

It is important to note that all presented methods usually generate a budget in a fixed interval $[a, b]$ and the actual interpretation of this budget is conferred to the user. For instance, HYPERBAND and SUCCESSIVEHALVING have been used very successfully to select the number of training epochs in neural networks. Consequently, multi-fidelity approximations can be used for many problem instances.

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 to abort the fitting after the first fold if the performance is significantly worse than the current incumbent (Maron & 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 (Domhan et al., 2015; Klein et al., 2017b). This allows discarding probably bad performing configurations without a complete training. By considering multiple

configurations in an iterative procedure simultaneously, the most promising configuration can be optimized in each step (Swersky et al., 2014).

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 drawn-out optimization periods (Huberman et al., 1997). By evaluating multiple problem instances in parallel, long running instances can be discarded early (Weisz et al., 2018; Li et al., 2020).

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 & Ghemawat, 2008). SCIKIT-LEARN (Pedregosa et al., 2011), which is used by most evaluated frameworks (see Section 8), already implements 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.

Most of the previously mentioned methods allow easy parallelization of single evaluations. Using grid search and random search, pipeline instances can be sampled independently of each other. Evolutionary algorithms allow a simultaneous evaluation of candidates in the same generation. However, SMBO is—as the name already implies—a sequential procedure.

SMBO procedures often contain a randomized component. Executing multiple SMBO instances with different random seeds allows a simple parallelization (Hutter et al., 2012). However, this simple approach often does not allow sharing knowledge between the different instances. Alternatively, the surrogate model M can be handled by a single *coordinator* while the evaluation of candidates is distributed to several *workers*. Pending candidate evaluations can be either ignored—if sampling a new candidate depends on a stochastic process (Bergstra et al., 2011; Kandasamy et al., 2018)— or imputed with a constant (Ginsbourger et al., 2010b) or simulated performance (Ginsbourger et al., 2010a; Snoek et al., 2012; Desautels et al., 2014). This way, new configurations can be sampled from an approximated posterior while preventing the evaluation of 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 experts without programming skills to create optimized ML workflows (USU Software AG, 2018; Baidu, 2018; RapidMiner, 2018).
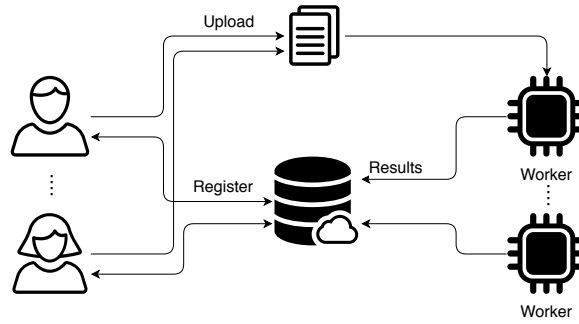
Figure 9: Components of an AutoML service (Swearingen et al., 2017).

7.4 Ensemble Learning

A well-known concept in ML is ensemble learning (Opitz & Maclin, 1999; Polikar, 2006; Rokach, 2010). 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 increased significantly. The cost of evaluating multiple ML models is often neglectable considering the performance improvements.

During the search of 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). Similarly, 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., 2018a).

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 previous layers (Wistuba et al., 2017; Khurana et al., 2018a; Chen et al., 2018).

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 (2019) provides a survey exclusively on meta-learning.

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

Search Space Refinements All presented CASH methods require an underlying search space definition. Often these search spaces are chosen arbitrarily 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 & Hutter, 2018; Probst et al., 2019) or identify promising regions (Wistuba et al., 2015b). Perrone et al. (2019) use transfer learning to automatically construct a minimal search space from the best configurations on related ML tasks.

Candidate Configuration Suggestion   Many AutoML procedures generate candidate configurations by selecting the configuration with the highest expected improvement. Meta-learning can be used as an additional criterion for selecting promising candidate configurations based on the predicted performance (e.g., Alia & Smith-Miles, 2006; Wistuba et al., 2015b; Nargesian et al., 2017) or ranking of the models (e.g., Sohn, 1999; Gama & Brazdil, 2000). Consequently, the risk of superfluous configuration evaluations is minimized.

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

Pipeline Structure   Meta-learning is also applicable for pipeline structure search. Feurer et al. (2015a) use meta-features to warm-start the pipeline synthesis. Using information on which preprocessing and model combination performs well, potentially better performing pipelines can be favored (Post et al., 2016; Bilalli et al., 2017; Schoenfeld et al., 2018). Gil et al. (2018) uses meta-features in the context of planning to select promising pipeline structures. Similarly, Drori et al. (2019) and Rakotoarison et al. (2019) use meta-features of the data set and pipeline candidate to predict the performance of the pipeline.

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

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

with T being the set of all known ML tasks, is necessary. Meta-data usually comprises properties of the previous task in combination with the used configuration and resulting model evaluations (Vanschoren, 2019).

A simple task-independent approach for ranking configurations is sorting $\mathbf{P}$ by performance. Configurations with higher performance are more favorable (Vanschoren, 2019). 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 performing well for a regression task may not be applicable to a classification problem.

An ML task t_j 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. The actual usage of $\vec{m}(t_j)$ highly depends on the meta-learning technique. For example, 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 then used similarly to task-independent meta-learning (Vanschoren, 2019).

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. Preferably, the frameworks cover a wide range of the methods presented in Section 3–7 without implementing the same approaches multiple times. Finally, all frameworks had to be open source.

Implementations of CASH algorithms are presented and analyzed in Section 8.1. 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. A reference to the source code of each framework is provided in Appendix A.

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.).

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

Grid Search   A custom implementation based on GridSearchCV from scikit-learn (Pedregosa et al., 2011) is used. GridSearchCV is extended to support algorithm selection via a distinct GridSearchCV instance for each ML algorithm. To ensure fair results, a mechanism for stopping the optimization after a fixed number of iterations has been added.

Random Search   Similar to grid search, a custom implementation of random search based on the scikit-learn implementation RandomizedSearchCV is used. RandomizedSearchCV is extended to support algorithm selection.

RoBO   RoBO (Klein et al., 2017a) is a generic framework for general purpose Bayesian optimization. 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 tuned automatically using Markov chain Monte Carlo sampling. Categorical hyperparameters are not supported. RoBO is evaluated in version 0.3.1.

BTB   BTB (Gustafson, 2018) combines multi-armed bandit learning with Gaussian processes. Categorical hyperparameters are selected via bandit learning and the remaining continuous hyperparameters are selected via Bayesian optimization. In the context of this work *upper confidence bound* is used as the policy. BTB is evaluated in version 0.2.5.

Hyperopt   hyperopt (Bergstra et al., 2011) is a CASH solver based on SMBO with TPEs as surrogate models. hyperopt is evaluated in version 0.2.

SMAC   SMAC (Hutter et al., 2011) was the first framework explicitly supporting categorical variables for configuration selection based on SMBO, making it especially suited for CASH. The performance of all previous configurations is modeled using random forest regression. SMAC automatically terminates single configuration evaluations after a fixed timespan. This way, very unfavorable configurations are discarded quickly without slowing the complete optimization down. SMAC is evaluated in version 0.10.0.

BOHB   BOHB (Falkner et al., 2018) combines Bayesian optimization with Hyperband (Li et al., 2018) for CASH optimization. A limitation of Hyperband is the random generation of the tested configurations. BOHB replaces this random selection by a SMBO procedure based on TPEs. For each function evaluation, BOHB passes the current budget and a configuration instance to the objective function. In the context of this evaluation, the budget is treated as the fraction of training data used for training. BOHB is evaluated in version 0.7.4.

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 is used. 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 AutoML frameworks capable of building complete ML pipelines based on the methods provided in Section 3, 5, and 6. For algorithm selection and HPO, implementations from Section 8.1 are used. A summary is available in Table 2.

Framework	CASH Solver	Structure	Ensem.	Cat.	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. It is listed whether ensemble learning (Ensem.), categorical input (Cat.), parallel evaluation of pipelines or a timeout for evaluations are supported (Time).

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.

TPOT TPOT (Olson & Moore, 2016; Olson et al., 2016) is a framework for building and tuning flexible classification and regression pipelines based on genetic programming. Regarding HPO, TPOT can only handle categorical parameters; similar to grid search all continuous hyperparameters have to be discretized. 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. TPOT supports basically all popular SCIKIT-LEARN preprocessing, classification and regression methods. It is evaluated in version 0.10.2.

Hyperopt-Sklearn HYPEROPT-SKLEARN or HPSKLEARN (Komer et al., 2014) is a framework for fitting classification and regression pipelines based on HYPEROPT. The pipeline structure is fixed to exactly one preprocessor and one classification or regression algorithm; all algorithms are based on SCIKIT-LEARN. HPSKLEARN only provides a thin wrapper around HYPEROPT by introducing the fixed pipeline structure and adding a configuration space definition. A parallelization of the configuration evaluation is not available. It supports only a rudimentary data preprocessing, namely principal component analysis (PCA), standard or min-max scaling and normalization. Additionally, the most popular SCIKIT-LEARN classification and regression methods are supported. HPSKLEARN is evaluated in version 0.0.3.

Auto-Sklearn AUTO-SKLEARN (Feurer et al., 2015a, 2018) is a tool for building classification and regression pipelines. All pipeline candidates have a semi-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. 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 and each evaluation is limited by a time budget. AUTO-SKLEARN uses meta-learning to initialize the optimization procedure. Additionally, ensemble learning is implemented by combining the best pipelines. AUTO-SKLEARN is evaluated in version 0.5.2.

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

ATM   ATM (Swearingen et al., 2017) is a collaborative service for building optimized classification pipelines based on BTB. Currently, ATM uses a simple pipeline structure with an optional PCA, an optional scaling followed by a tunable classification algorithm. All algorithms are based on SCIKIT-LEARN and popular classification algorithms are supported.

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. Yet, currently this potential is not utilized. ATM is evaluated in version 0.2.2.

H2O AutoML   H2O (H2O.ai, 2019) is a distributed ML framework to assist data scientists. In the context of this paper, only the H2O AUTOML component is considered. H2O AUTOML is able to select and tune a classification algorithm without preprocessing automatically. Available algorithms are tested in a fixed order with either expert-defined or via randomized grid-search selected hyperparameters. In the end, the best performing configurations are aggregated to create an ensemble. 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 frameworks. 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 discussed. Finally, all selected frameworks are evaluated empirically on real data.

9.1 Comparability of Results

A reliable and fair comparison of different AutoML algorithms and frameworks is 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 implemen-

tation details, like refitting a model on the complete data set after cross-validation, many design decisions can skew the performance comparison heavily. 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. It 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 or 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 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.

9.2 Benchmarking Methodology

All experiments are 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 uses 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 are conducted:

---

2. Available at http://automl.chalearn.org/.
3. For more information see https://cloud.google.com/compute/docs/machine-types.

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

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

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

2. CASH solvers (see Section 9.5.1) are limited to exactly 325 iterations. Preliminary evaluations have shown that all algorithms 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 with three folds passed to the optimizer and using the last fold to calculate a test-performance. As loss function, the accuracy

$$\mathcal{L}_{\text{Acc}}(\hat{y}, y) = \frac{1}{|y|} \sum_{i=1}^{|y|} \mathbb{1}(\hat{y}_i = y_i) \tag{6}$$

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 with three folds passed to the AutoML framework[4] and using the last fold to calculate a test-performance. As loss function, the accuracy in Equation (6) is used.

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. Frameworks supporting parallelization are configured to use eight threads. Furthermore, frameworks supporting memory limits are configured to use at most 4096 MB memory per thread. The source code used for the benchmarks is available online[5].

For the third experiment, we also tested cut-off timeouts of 4 and 8 hours on ten randomly selected data sets. The performance after 4 or even 8 hours did only marginally improve in comparison to 1 hour and is therefore not further considered.

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

---

4. Internally, the AutoML frameworks may implement different methods to prevent overfitting, e.g., a nested cross-validation or a hold-out data set.
5. Available at `https://github.com/Ennosigaeon/automl_benchmark`.

constant time. 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. A short evaluation of the presented CASH algorithms on selected synthetic test functions is given in Appendix B.

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 an AutoML algorithm, the time for fitting a single model can become prohibitively large. In order to lower the turnaround time for testing a single configuration significantly, empirical performance models (EPMs) have been introduced (Eggensperger et al., 2015, 2018).

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 reduced significantly. 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, due to the quasi exhaustive analysis of the configuration space, EPMs suffer heavily 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 previously introduced methods for performance evaluations only consider selecting and tuning a modeling algorithm. Data cleaning and feature engineering are ignored completely 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 evaluation overhead, 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 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[6] (Bischl et al., 2017), OpenML-CC18[7] (Bischl et al., 2019) and AutoML Benchmark[8] (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. 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 data sets with some basic meta-features is provided in Appendix C. All CASH algorithm and most AutoML frameworks do not support categorical features. Therefore, categorical features of all data sets are transformed using one hot encoding. Furthermore, data sets are shuffled to remove potential impacts of ordered data.

9.5.1 CASH Algorithms

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 3 and Appendix D—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.

Algorithm	$\#\lambda$	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 3: 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 ($\#\lambda$) is listed.

---

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

7. Available at `https://www.openml.org/s/99`.

8. Available at `https://www.openml.org/s/218`.

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 15 in Appendix E contains the raw results of the evaluation. It reports the average accuracies 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 an interval between zero and one to obtain data set independent evaluations. Zero represents the performance of the dummy classifier 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.
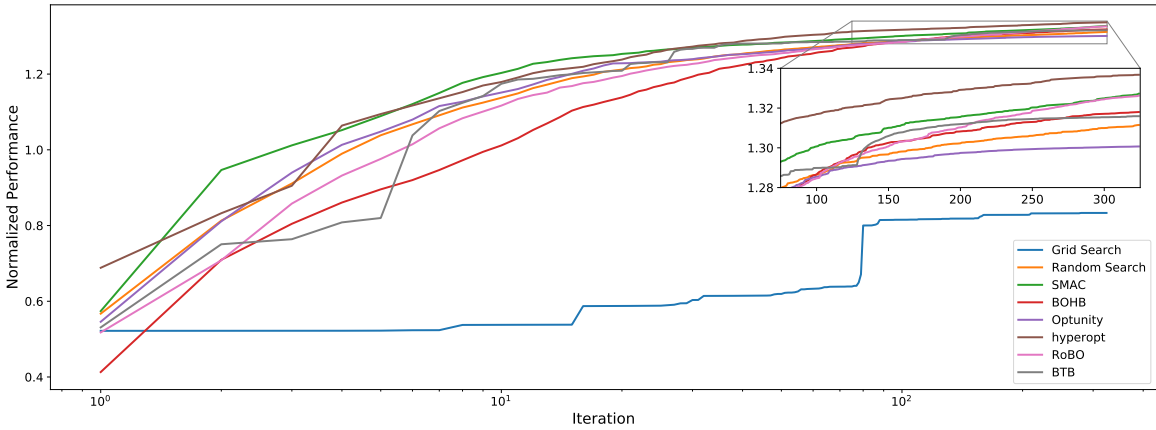
Figure 10: Normalized performance of the incumbent per iteration. Results are averaged over all data sets and 10 repetitions.

	Grid	Random	SMAC	BOHB	OPTUNITY	HYPEROPT	RoBO	BTB
Rep.	0.0656	0.0428	0.0395	0.0414	0.0514	0.0483	0.0421	0.0535
Data Set	0.7655	1.1004	1.1420	1.1478	1.0732	1.1206	1.1334	1.1302

Table 4: Standard deviation of the normalized performance of the final incumbent averaged over ten repetitions (Rep.) and all data sets (Data Set).

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. Table 4 contains the standard deviation of the normalized performance of the final incumbent after the optimization. Values averaged over ten repetitions and all data sets are shown. It is apparent that the normalized performance heavily depends on the used data set.
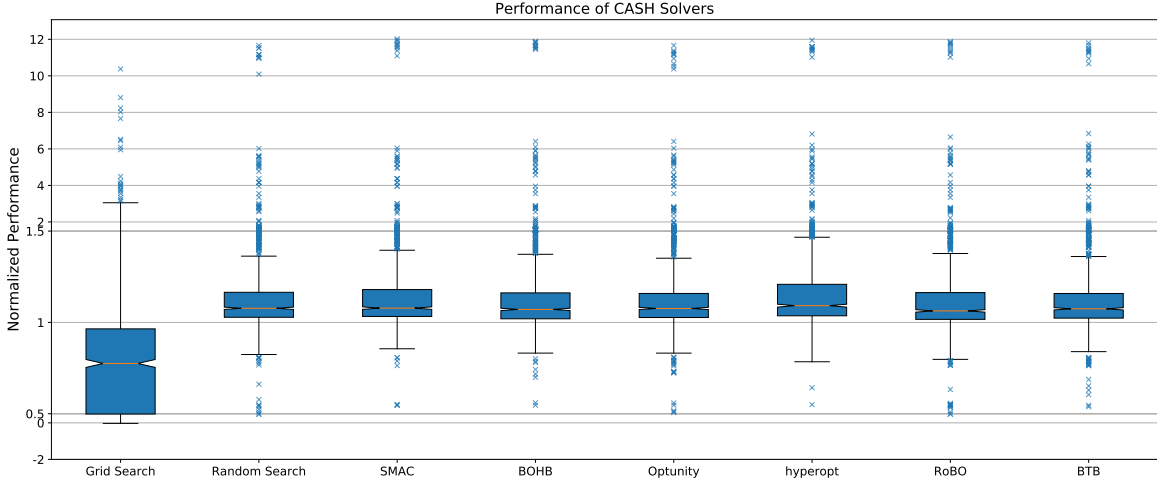
Figure 11: Normalized performance of the final incumbent per CASH solvers. For better readability, performances between 0.5 and 1.5 are stretched out.

A pair-wise comparison of the performances of the final incumbent is displayed in Table 5. It is apparent that HYPEROPT outperforms all other optimizers and grid search is basically always outperformed. Yet, a more detailed comparison of performances, provided in Figure 20 in Appendix E, reveals that absolute performance differences are small.

	Grid	Random	SMAC	BOHB	OPTUNITY	HYPEROPT	RoBO	BTB
Grid	–	0.0263	0.0175	0.0175	0.0263	0.0175	0.0175	0.0263
Random	0.9561	—	0.3771	0.6403	0.5175	0.0614	0.5964	0.5614
SMAC	0.9649	0.5614	—	0.8508	0.6228	0.2192	0.7105	0.6403
BOHB	0.9649	0.2807	0.0877	—	0.3596	0.0877	0.4385	0.3859
OPTUNITY	0.9561	0.4385	0.3245	0.5877	—	0.1403	0.5263	0.5087
HYPEROPT	0.9649	0.8684	0.7368	0.8684	0.8157	—	0.7894	0.8947
RoBO	0.9649	0.3596	0.2456	0.5087	0.4385	0.1491	—	0.3947
BTB	0.9561	0.3859	0.3070	0.5701	0.4385	0.0614	0.5614	—
Avg. Rank	7.7280	3.9210	3.0964	5.0438	4.2192	1.7368	4.6403	4.4122

Table 5: Fraction of data sets on which the CASH solvers in each row performed better than the framework in each column. As CASH solvers can yield identical performances, the according fractions do not have to add up to 1. Additionally, the rank of each CASH solver is given.

Figure 12 shows the raw scores for each CASH framework over 10 repetitions for 16 data sets. Those data sets were selected as they show the highest deviation of the scores over ten repetitions. The remaining data sets yielded very consistent results. We do not know which data set properties are responsible for the unstable results.
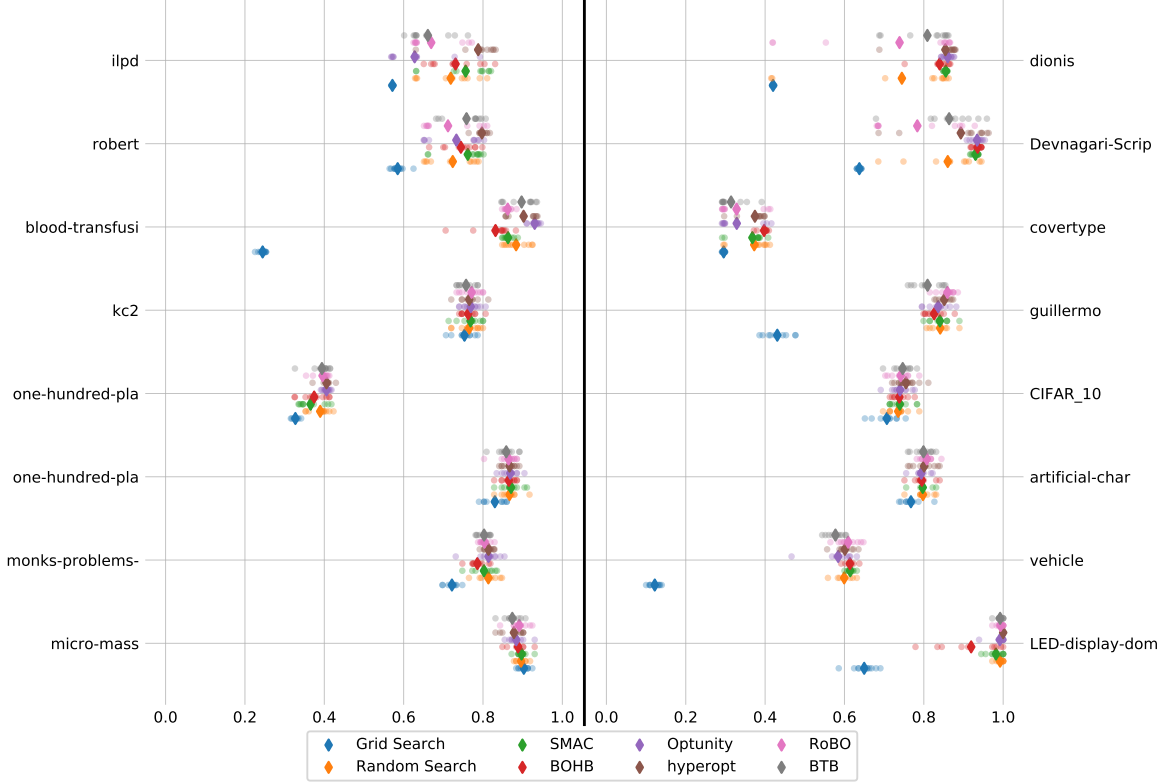
Figure 12: Raw and averaged accuracy of all CASH solvers on selected data sets.

Next, we examine the similarity of the proposed configurations per data set. Therefore, numerical hyperparameters are normalized by their according search space, categorical hyperparameters are not transformed. We decided to only compare configurations having the same classification algorithm. For each classification algorithm, all configuration vectors are aggregated using mean shift clustering (Fukunaga & Hostetler, 1975) with a bandwidth $h = 0.25$. To account for the mixed-type vector representations, the Gower distance (Gower, 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 13 shows the silhouette coefficient versus number of instances per cluster. Displayed are clusters of all configurations aggregated 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
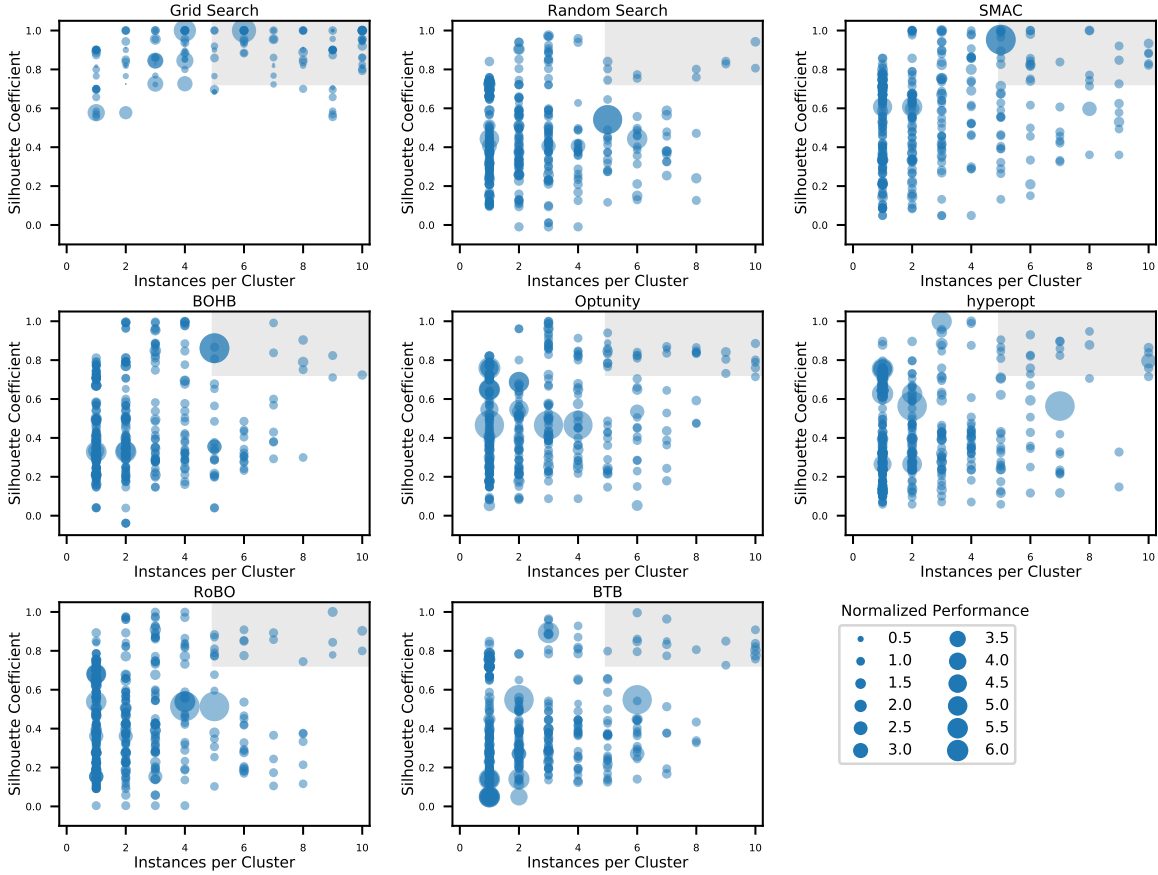
Figure 13: Similarity of configurations versus number of instances per cluster. Each marker represents the similarity of configurations for a single data set and single classification algorithm. The marker size indicates the normalized accuracy (larger equals higher accuracy). Clusters in the highlighted area are considered to contain similar configurations. Each subplot considers only configurations yielded by the stated CASH algorithm.

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.

Finally, we examine the known tendency of AutoML tools to overfit (Fabris & Freitas, 2019). In Figure 14, an estimate of the overfitting tendency of the different CASH solvers is given. Displayed are the differences between the accuracy on the training and test data set. It is apparent that on average, all evaluated methods—except grid search—have a similar tendency to overfit. For single instances, all CASH methods, again with the exception of grid search, suffer heavily from overfitting.

Figure 14: Overfit estimation between the learning and testing data set. Displayed are the raw differences between the accuracy scores. Larger values indicate higher overfitting.

9.5.2 AutoML Frameworks

Next, 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 and OpenML-CC18 suites unable to be processed by CASH procedures—namely data sets containing missing values—are selected. The final list of all 73 selected data sets is provided in Table 16 in Appendix E.

ATM does not provide the possibility to abort configuration evaluations after a fixed time and therefore often exceeds the total time budget. To enforce the time budget, all configuration evaluations are manually aborted after 1.25 hours. Random Search uses auto-sklearn with a random configuration generation. Meta-learning and ensemble support are deactivated. As hyperopt-sklearn does not support parallelization, only single-threaded evaluations of configurations are used. Furthermore, hyperopt-sklearn was manually extended to support a time budget instead of number of iterations. The remaining optimizers and all unmentioned parameters are used with their default parameters.

Table 16 in Appendix E contains the raw results of the evaluation. The average accuracy over all trials per data set is reported. In contrast to the CASH algorithms, the AutoML frameworks struggle with various data sets. ATM drops samples with missing values in the training set. 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 crashes with a segmentation fault. For multiple data sets TPOT times out after first generation. Consequently, only random search without genetic programming is performed. Data sets 40923, 41165 and 41167 time out consistently with no result. 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 crashes 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.
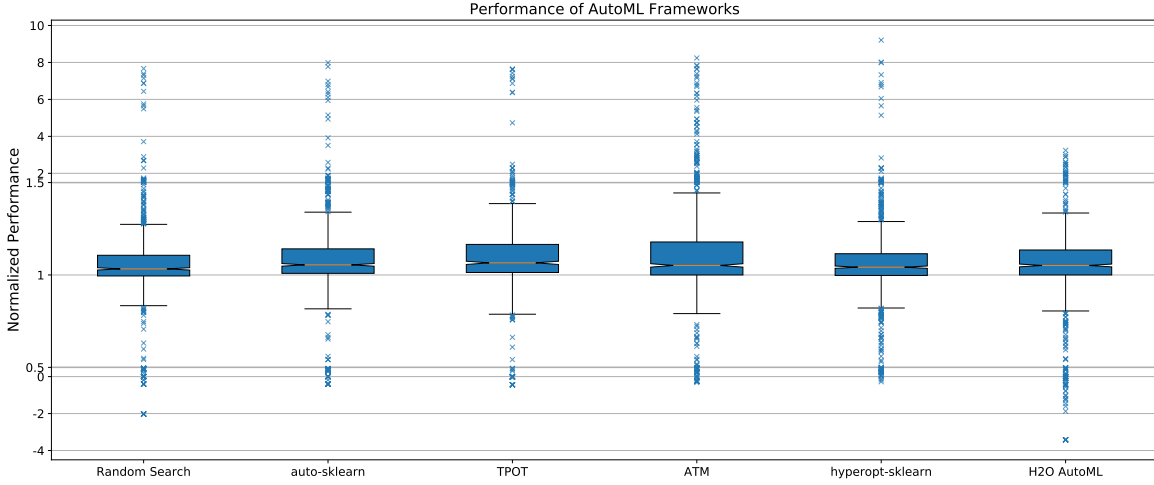
Figure 15: Normalized performance of the final pipeline per AutoML framework. For better readability, performances between 0.5 and 1.5 are stretched out.

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. Table 6 compares all framework pairs and lists the average rank for each framework. It is apparent that TPOT outperforms the most frameworks averaged over all data sets. A detailed pairwise comparison including the absolute performance differences is provided in Figure 20 in Appendix E.

	TPOT	HPSKLEARN	AUTO-SKLEARN	Random	ATM	H2O
TPOT	—	0.7571	0.6086	0.8529	0.6000	0.5000
HPSKLEARN	0.2285	—	0.2816	0.5571	0.4117	0.2898
AUTO-SKLEARN	0.3623	0.7042	—	0.8000	0.4848	0.5294
Random	0.1323	0.4428	0.2000	—	0.3846	0.3283
ATM	0.3692	0.5735	0.4848	0.6153	—	0.4687
H2O	0.4705	0.7101	0.4558	0.6716	0.5156	—
Avg. Rank	2.6027	4.0410	2.9863	4.4109	3.4931	3.1643

Table 6: Fraction of data sets on which the framework in each row performed better than the framework in each column. As frameworks can yield identical performances, the according fractions do not have to add up to 1. Additionally, the rank of each framework averaged over all frameworks is given.
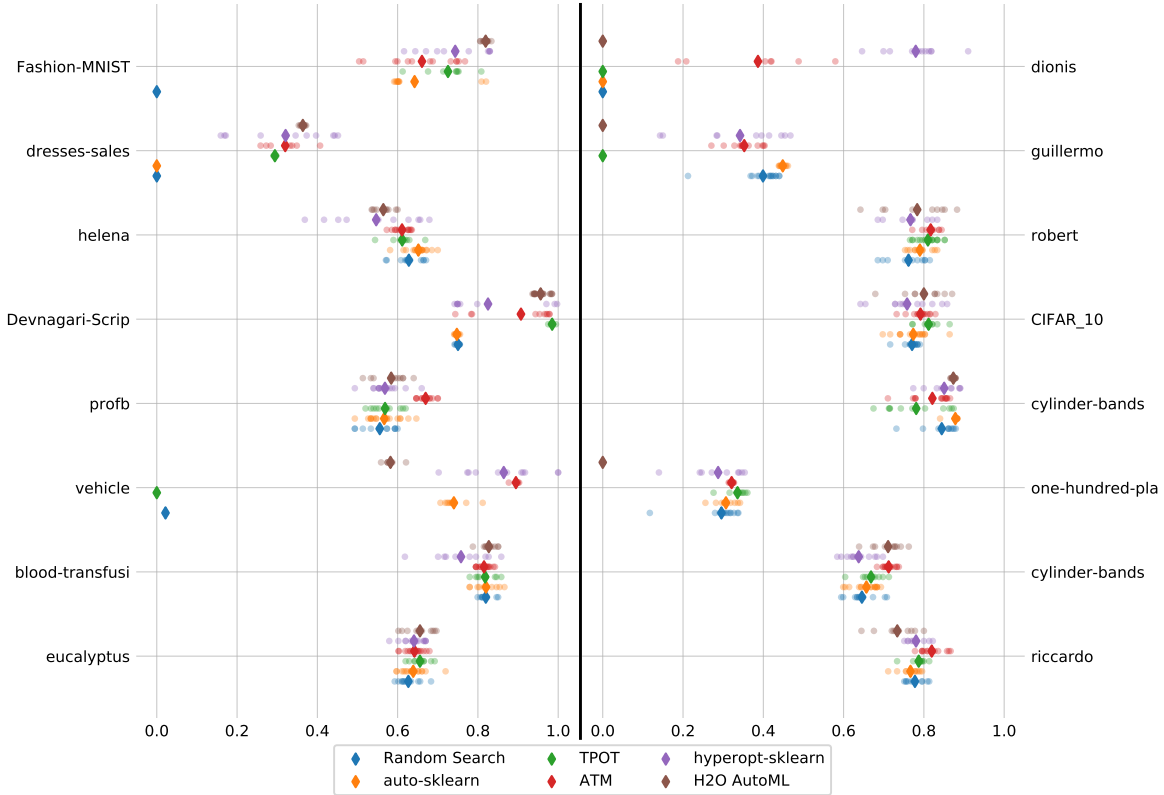
Figure 16: Raw and averaged accuracy of all AutoML frameworks on selected data sets.

	TPOT	HPSKLEARN	AUTO-SKLEARN	Random	ATM	H2O
Rep.	0.0761	0.1508	0.0843	0.0955	0.0963	0.0993
Data Set	0.7343	0.7004	0.6772	0.6956	0.8938	0.2526

Table 7: Standard deviation of the normalized performance of the final pipeline averaged over ten repetitions (Rep.) and all data sets (Data Set).

Figure 16 shows raw scores for each AutoML framework over ten trials for 16 data sets. Those data sets were selected as they show the highest deviation of the scores over the ten trials. About 50% of all 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. Table 7 contains the standard deviation of the normalized performance of the final pipeline after the optimization. Shown are averaged values over ten repetitions and all data sets. In comparison with the CASH solvers, the stability within ten iterations has decreased while the stability across data sets has increased.

Figure 17 shows an estimate of the test-training overfit for all evaluated frameworks. In general, the AutoML frameworks, especially random search and AUTO-SKLEARN, appear to be more prone to overfitting than CASH solvers. All tested frameworks overfit strongly for single instances.
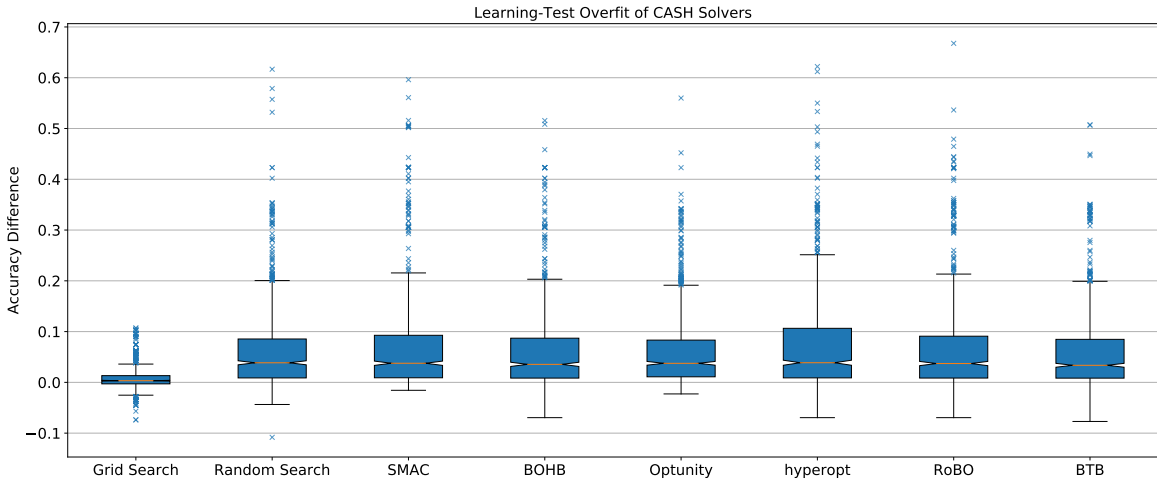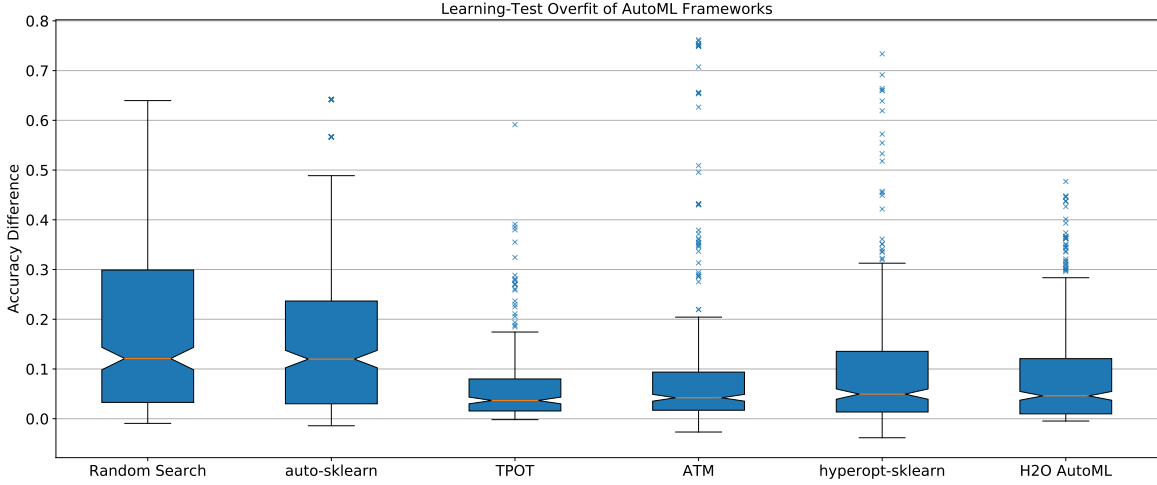
Figure 17: Overfit estimation between the learning and testing data set. Displayed are the raw differences between the accuracy scores. Larger values indicate higher overfitting.

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

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 sub-graph 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 expressed by the Levenshtein ratio (Levenshtein, 1966; Ratcliff & Metzener, 1988). Table 8 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 themselves. This can be explained by the long (semi-)fixed pipeline structure. All other AutoML frameworks yield very low similarity ratios. This can be explained partially by the different search spaces, i.e., the AutoML frameworks do not support identical base algorithms. Therefore, we also consider a generalized representation of the ML pipelines, e.g., replacing all classification algorithms with an identical symbol.

1	DecisionTree	8	StandardScaler	15	PCA	22	VarianceThreshold	29	MyDummy	36	FastICA
2	ExtraTrees	9	Normalizer	16	SelectPercentile	23	SVC	30	SGD	37	SelectFromModel
3	PolynomialFeatures	10	GaussianNB	17	SelectFwe	24	AdaBoost	31	QuantileTransformer	38	SelectRates
4	MaxAbsScaler	11	MinMaxScaler	18	RobustScaler	25	DeepLearning	32	Balancing	39	RandomKitchenSinks
5	GradientBoosting	12	RandomForest	19	BernoulliNB	26	GeneralizedLinear	33	QDA	40	Nystroem
6	LinearSVC	13	KNeighbors	20	MultinomialNB	27	OneHotEncoder	34	FeatureAgglomeration	41	KernelPCA
7	LogisticRegression	14	XGB	21	ZeroCount	28	Imputation	35	RandomTreesEmbedding	42	LDA

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.

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

Table 9 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.

9.5.3 Comparison with Human Experts

Finally, all AutoML frameworks are compared with human experts. Unfortunately, it is not possible to reuse the same data sets, as human evaluations for those data sets are not available. Instead, we decided to use two publicly available data sets from kaggle, namely *Otto Group Product Classification Challenge*[9] and *Santander Customer Satisfaction*[10]. Even though the evaluation of just two data sets provides only limited generalization, it can still be used to get a feeling for the competitiveness of AutoML tools with human experts.

The experimental setup from Section 9.5.2 is reused. Only the loss function is adapted to reflect the loss function used by the two challenges—*logarithmic loss* for *Otto* and *ROC AUC* for *Santander*. If any framework does not support the respective loss function, we continued to use accuracy.

	Otto			*Santander*		
	Validation	Test	Ranking	Validation	Test	Ranking
Human	—	0.38055	—	—	0.84532	—
TPOT	0.81066	1.05085	0.7908	0.83279	0.83100	0.6827
hpsklearn	0.81177	0.58701	0.6216	0.66170	0.64493	0.8789
auto-sklearn	0.55469	0.55081	0.5155	0.83547	0.83346	0.6543
Random	0.88702	0.89943	0.7777	0.82806	0.82427	0.7235
ATM	0.74912	2.43115	0.8459	0.68721	0.69043	0.8653
H2O	0.45523	0.49628	0.3774	0.83406	0.83829	0.5329

Table 10: Comparison with human experts for two data sets. Displayed are the validation and test score. Additionally, the fraction of human submissions that have yielded better results is given (Ranking). For *Otto* smaller validation and test values are better while for *Santander* higher values are better.

Table 10 compares all AutoML frameworks with the best human performance. For both data sets, all algorithms are able to achieve mediocre results that are outperformed by human experts clearly. A detailed look at the leaderboard reveals that human experts required on average 8.57 hours to refine their initial pipeline to outperform the best AutoML framework. Obviously, this duration does not incorporate the time spend to craft the initial solution. Considering that all frameworks spend only one hour, the results are still remarkable.

---

9. Available at `https://www.kaggle.com/c/otto-group-product-classification-challenge`.
10. Available at `https://www.kaggle.com/c/santander-customer-satisfaction`.

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 similarly. Surprisingly, random search did not perform worse than the other algorithms. The performance differences of the final configurations are not significant for most data sets with 67.18% of all configurations not being significantly worse than the best result. Mean absolute differences are less than 1.9% accuracy per data set. Consequently, a ranking of CASH algorithms on pure performance measures is not reasonable. Other aspects like scalability or method overhead should also be considered.

On average, all AutoML frameworks appear to perform quite similarly with a maximum performance difference of only 2.2% and three frameworks yielding no significantly worse results than the best framework. Yet, the global average conceals that for each individual data set the performance differs by 6.7% accuracy averaged over all frameworks. Only 43.61% of the final pipelines are not significantly worse than the best pipeline. In addition, the CASH algorithms performed better than the AutoML frameworks on 48% of the shared data sets (see Table 15 and 16 in Appendix E). This is also a surprising result as each CASH algorithm spends on average only 12 minutes optimizing a single data set in contrast to the 1 hour of AutoML frameworks. Possible explanations for both observations could be the significantly larger search spaces of AutoML frameworks, a smaller number of evaluated configurations due to internal overhead, e.g., cross-validations, or the tendency of AutoML frameworks to overfit stronger than CASH solvers. Further evaluations are necessary to explain this behavior.

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) or P4ML (Gil et al., 2018), in combination with longer optimization periods is desirable to understand the capabilities of creating adaptable pipelines better.

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. Data scientists usually spend 60–80% of their 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 combine predefined operators with features naively. For building flexible pipelines, currently only a few different approaches

have been proposed. Further research in any of these three areas can improve the overall performance of an automatically created ML pipeline highly.

So far, researchers have focused on a single point of the pipeline creation process. Combining flexibly structured pipelines with automatic feature engineering and sophisticated CASH methods has the potential to beat the frameworks currently available. 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 to build complete pipelines with every single component optimized automatically.

AutoML aims to automate the creation of an ML pipeline completely to enable domain experts to use ML. Except very few publications (e.g., Friedman & 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 reduced significantly.

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

Even though methods like *feature attribution* (Google LLC, 2019) or *rule-extraction* (Alaa & Van Der Schaar, 2018) have already been used in combination with AutoML, the black-box problem still prevails. 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. Domain experts would be able to guide model creation by their experience. Further research in this area may lead to more profound models depicting the real-world dependencies closer. Simultaneously, the domain expert would have the chance to understand the reasoning of the ML model better. This could increase the acceptance of the proposed pipeline.

AutoML frameworks usually introduce their own hyperparameters that can be tuned. 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 & 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 conduct an ML project successfully, all stages in the CRISP-DM should be considered. To make AutoML truly available to 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 introduced and summarized theoretically. 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 ten years, it has received a lot of attention from research, enterprises and the media. Current state-of-the-art frameworks enable domain experts to build reasonably 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 future.

Acknowledgments

Appendix A. Framework Source Code

Table 11 lists the repositories of all evaluated open-source AutoML tools. Some methods are still under active development and may differ significantly from the evaluated versions.

Algorithm	Type	Source Code
Custom	Both	`https://github.com/Ennosigaeon/automl_benchmark`
RoBO	CASH	`https://github.com/automl/RoBO`
BTB	CASH	`https://github.com/HDI-Project/BTB`
HYPEROPT	CASH	`https://github.com/hyperopt/hyperopt`
SMAC	CASH	`https://github.com/automl/SMAC3`
BOHB	CASH	`https://github.com/automl/HpBandSter`
OPTUNITY	CASH	`https://github.com/claesenm/optunity`
TPOT	AutoML	`https://github.com/EpistasisLab/tpot`
HPSKLEARN	AutoML	`https://github.com/hyperopt/hyperopt-sklearn`
AUTO-SKLEARN	AutoML	`https://github.com/automl/auto-sklearn`
ATM	AutoML	`https://github.com/HDI-Project/ATM`
H2O AUTOML	AutoML	`https://github.com/h2oai/h2o-3`

Table 11: Source code repositories for all used CASH and AutoML frameworks.

Appendix B. Synthetic Test Functions

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 12 contains the performance of each algorithm after the completed optimization. Over all benchmarks, RoBO was able to consistently outperform or yield equivalent results compared to all competitors.

Benchmark	Grid	Random	RoBO	BTB	HYPEROPT	SMAC	BOHB	OPTUNITY
Levy	0.00089	0.00102	**0.00000**	0.19588	0.00010	0.00058	0.02430	0.00013
Branin	0.24665	0.28982	**0.00065**	<u>0.00077</u>	0.05011	0.10191	0.39143	0.03356
Hartmann6	1.04844	0.66960	**0.06575**	0.27107	0.44905	0.27262	0.35435	<u>0.22289</u>
Rosenbrock10	9.00000	45.8354	**4.43552**	19.4919	22.4746	38.1581	34.4457	36.3984
Camelback	0.94443	0.45722	<u>0.02871</u>	<u>0.07745</u>	0.07594	0.18440	0.38247	**0.01754**

Table 12: Results of all tested CASH solvers after 100 iterations. For each synthetic benchmark the mean performance over 10 trials is reported. Bold face represents the best mean value for each benchmark. Results not significantly worse than the best result—according to a Wilcoxon signed-rank test—are underlined.

Appendix C. 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

Name	Id	Classes	Samples	Numeric	Categorical	Missing values	Incomp. Samples	Minority %
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 13: 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 sample, 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 D. 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 14: 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 E. 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	<u>0.9950</u>	0.9948	0.9944	**0.9952**	<u>0.9948</u>	<u>0.9945</u>	<u>0.9941</u>
1038	0.5014	0.9065	0.8012	0.9376	0.9375	0.9335	<u>0.9423</u>	**0.9516**	0.9302	0.9418
1043	0.6270	0.8297	0.7879	<u>0.8521</u>	<u>0.8524</u>	<u>0.8500</u>	<u>0.8517</u>	<u>0.8565</u>	0.8486	**0.8568**
1046	0.5582	0.9492	0.9353	<u>0.9583</u>	<u>0.9580</u>	0.9533	<u>0.9583</u>	**0.9605**	0.9538	0.9555
1049	0.7779	0.8975	0.8747	<u>0.9178</u>	<u>0.9185</u>	<u>0.9153</u>	<u>0.9187</u>	**0.9235**	<u>0.9121</u>	<u>0.9151</u>
1050	0.8158	0.8893	0.8663	<u>0.9053</u>	<u>0.9068</u>	<u>0.9053</u>	<u>0.9053</u>	**0.9100**	0.8983	<u>0.9051</u>
1063	0.6828	0.8127	0.8299	<u>0.8669</u>	**0.8707**	<u>0.8650</u>	<u>0.8688</u>	<u>0.8669</u>	0.8643	0.8586
1067$^+$	0.7409	0.8504	0.8509	<u>0.8649</u>	<u>0.8660</u>	<u>0.8621</u>	<u>0.8640</u>	<u>0.8687</u>	<u>0.8657</u>	**0.8727**
1068	0.8670	<u>0.9330</u>	0.9261	<u>0.9396</u>	0.9402	0.9363	0.9381	<u>0.9432</u>	**0.9438**	0.9372
1120	0.5455	0.8664	0.6491	<u>0.8790</u>	<u>0.8797</u>	0.8766	0.8802	**0.8819**	0.8714	<u>0.8794</u>
1169$^-$	0.5060	0.6144	0.5545	<u>0.6650</u>	<u>0.6655</u>	<u>0.6635</u>	0.6639	**0.6655**	0.6627	<u>0.6627</u>
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	<u>0.9079</u>	<u>0.9078</u>	0.9070	**0.9084**	<u>0.9071</u>	0.9052	0.9044
1462	0.5056	0.9925	0.8451	**1.0000**	**1.0000**	**1.0000**	<u>0.9995</u>	**1.0000**	**1.0000**	<u>0.9995</u>
1464$^-$	0.6418	0.7329	0.7676	<u>0.7978</u>	<u>0.7973</u>	<u>0.7951</u>	<u>0.7938</u>	<u>0.8009</u>	**0.8076**	<u>0.7991</u>
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	<u>0.9179</u>	<u>0.9198</u>	<u>0.9167</u>	<u>0.9173</u>	**0.9284**	<u>0.9204</u>	<u>0.9247</u>
1468$^+$	0.1139	0.8985	<u>0.9586</u>	0.9571	**0.9630**	<u>0.9614</u>	0.9562	<u>0.9599</u>	<u>0.9617</u>	<u>0.9537</u>
1471	0.5074	0.8915	0.5519	0.9522	**0.9741**	<u>0.9729</u>	0.9541	<u>0.9726</u>	0.9414	0.9459
1475$^+$	0.2441	0.5822	0.3670	<u>0.6082</u>	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	<u>0.9908</u>	<u>0.9896</u>	0.9857	**0.9916**	0.9873	0.9885
1479	0.5074	0.5459	0.7857	0.9354	<u>0.9558</u>	**0.9566**	0.9321	<u>0.9492</u>	<u>0.9511</u>	0.9431
1480	0.5909	0.7034	0.7069	<u>0.7354</u>	<u>0.7394</u>	<u>0.7383</u>	<u>0.7400</u>	**0.7550**	<u>0.7417</u>	<u>0.7469</u>
1485	0.4991	0.6191	0.5922	<u>0.8351</u>	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	<u>0.9435</u>	0.9351	<u>0.9460</u>	<u>0.9468</u>	<u>0.9447</u>	<u>0.9466</u>	<u>0.9482</u>	**0.9501**	<u>0.9470</u>
1489$^-$	0.5838	0.8873	0.7588	<u>0.9004</u>	<u>0.9002</u>	<u>0.8946</u>	<u>0.8986</u>	**0.9028**	<u>0.8990</u>	0.8949
1491	0.0100	0.6177	**0.8252**	0.8096	0.8144	0.7929	0.8117	0.8094	<u>0.8100</u>	0.8010
1492$^-$	0.0100	0.5135	0.1219	0.5994	**0.6146**	<u>0.6137</u>	0.5842	<u>0.6012</u>	<u>0.6094</u>	0.5773
1493	0.0104	0.6412	0.7217	<u>0.8135</u>	<u>0.8025</u>	0.7858	**0.8138**	<u>0.8138</u>	<u>0.8037</u>	<u>0.8027</u>
1494	0.5634	0.8492	0.7924	<u>0.8814</u>	**0.8893**	<u>0.8795</u>	0.8823	<u>0.8849</u>	0.8760	<u>0.8804</u>
1497	0.3356	0.9908	0.5913	<u>0.9979</u>	<u>0.9971</u>	0.9962	<u>0.9977</u>	**0.9983**	0.9966	<u>0.9975</u>
1501	0.1008	0.8690	0.8559	<u>0.9475</u>	<u>0.9513</u>	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	<u>0.9713</u>	<u>0.9713</u>	<u>0.9719</u>	**0.9749**	<u>0.9719</u>	<u>0.9731</u>	<u>0.9737</u>
1515	0.0599	0.7971	**0.9029**	<u>0.8959</u>	<u>0.8971</u>	<u>0.8884</u>	0.8837	0.8779	0.8913	0.8738
1570	0.8988	0.9814	0.9450	<u>0.9857</u>	<u>0.9863</u>	<u>0.9853</u>	<u>0.9841</u>	**0.9864**	<u>0.9848</u>	<u>0.9851</u>
1596$^-$	0.3771	**0.9388**	0.6375	0.8603	0.9303	<u>0.9356</u>	<u>0.9344</u>	<u>0.8933</u>	0.7836	0.8638
4134$^+$	0.5109	0.7586	0.6604	<u>0.7967</u>	<u>0.8017</u>	<u>0.7956</u>	<u>0.7937</u>	**0.8058**	0.7942	<u>0.7969</u>
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**	<u>0.9488</u>	0.9475
4534$^+$	0.5062	0.9696	0.9097	<u>0.9695</u>	<u>0.9701</u>	<u>0.9692</u>	<u>0.9712</u>	**0.9724**	0.9658	0.9694
4534$^+$	0.5018	0.9688	0.9115	<u>0.9708</u>	<u>0.9698</u>	<u>0.9682</u>	<u>0.9711</u>	**0.9726**	0.9646	<u>0.9699</u>
4538$^-$	0.2374	0.5936	0.3597	0.6505	**0.6876**	0.6674	0.6405	<u>0.6755</u>	0.6349	0.6469
23517$^+$	0.4987	0.5031	0.5140	<u>0.5220</u>	<u>0.5225</u>	<u>0.5230</u>	<u>0.5221</u>	<u>0.5215</u>	<u>0.5236</u>	**0.5236**
40496	0.0947	0.7000	<u>0.7533</u>	<u>0.7653</u>	<u>0.7687</u>	<u>0.7627</u>	<u>0.7693</u>	<u>0.7653</u>	**0.7713**	<u>0.7573</u>
40499	0.0888	0.9622	0.2067	<u>0.9981</u>	0.9981	0.9977	<u>0.9976</u>	**0.9988**	<u>0.9979</u>	<u>0.9981</u>
40509	0.5145	0.8667	<u>0.8831</u>	<u>0.8937</u>	<u>0.8932</u>	<u>0.8903</u>	<u>0.8932</u>	**0.8947**	<u>0.8903</u>	<u>0.8889</u>
40668$^-$	0.5035	0.7868	0.6364	0.8012	0.8023	0.7968	0.7986	**0.8084**	0.8027	<u>0.8019</u>
40670$^-$	0.3855	0.9182	0.9449	<u>0.9635</u>	<u>0.9621</u>	<u>0.9616</u>	<u>0.9655</u>	**0.9656**	0.9552	**0.9656**
40685$^-$	0.6439	<u>0.9997</u>	0.8191	0.9995	<u>0.9997</u>	0.9995	0.9996	**0.9998**	0.9996	0.9994
40701$^+$	0.7529	0.9476	0.8601	<u>0.9591</u>	<u>0.9603</u>	0.9585	<u>0.9592</u>	**0.9618**	0.9531	<u>0.9561</u>
40923$^-$	0.0213	<u>0.7779</u>	0.5717	<u>0.7187</u>	<u>0.7562</u>	<u>0.7308</u>	0.6277	**0.7879**	0.6694	0.6610
40927$^+$	0.0994	0.3510	0.2956	<u>0.3726</u>	0.3680	**0.3974**	0.3285	0.3744	0.3282	0.3142
40975$^+$	0.5395	0.9563	0.7597	<u>0.9881</u>	<u>0.9911</u>	0.9723	<u>0.9956</u>	**0.9963**	0.9873	<u>0.9913</u>
40978$^+$	0.7520	0.9735	0.9685	<u>0.9780</u>	<u>0.9778</u>	<u>0.9754</u>	<u>0.9771</u>	**0.9792**	0.9738	<u>0.9744</u>
40979$^+$	0.0962	0.9522	0.9185	<u>0.9822</u>	<u>0.9825</u>	<u>0.9810</u>	<u>0.9823</u>	**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 15: 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. Results not significantly worse than the best result—according to a Wilcoxon signed-rank test—are underlined. On data sets marked by [+] and [−], CASH solvers performed better and worse, respectively, than AutoML frameworks.

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	<u>0.89716</u>	**0.90450**	<u>0.89963</u>	<u>0.89273</u>	<u>0.89205</u>
1492^+	0.00687	0.51333	<u>0.62795</u>	**0.65172**	<u>0.61146</u>	<u>0.61097</u>	<u>0.54667</u>	<u>0.56435</u>
1590	0.63379	0.85021	<u>0.87013</u>	<u>0.86938</u>	**0.87089**	0.85448	<u>0.86727</u>	0.86656
1596^+	0.37644	0.93818	0.89143	**0.96395**	<u>0.94542</u>	0.66390	0.95227	0.92908
4134^-	0.50462	0.76314	0.77762	<u>0.78890</u>	**0.80249**	0.77087	0.77798	<u>0.80044</u>
4135^+	0.88895	0.94491	0.94444	0.94761	0.94891	0.94606	0.94750	**0.95114**
4534^-	0.50612	<u>0.96847</u>	0.96244	0.96590	<u>0.96913</u>	0.96464	<u>0.96964</u>	**0.97160**
4538^+	0.23130	0.59207	0.65004	0.67733	0.67586	0.66217	0.67272	**0.70165**
4550	0.12346	0.99414	<u>0.99907</u>	**1.00000**	**1.00000**	**1.00000**	<u>0.99983</u>	**1.00000**
6332	0.52407	0.73951	0.76173	0.79012	<u>0.81009</u>	**0.81701**	0.76667	<u>0.78333</u>
6332	0.49877	0.76481	0.77058	0.77353	**0.81173**	<u>0.79155</u>	0.75823	<u>0.80000</u>
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**	<u>0.72031</u>	0.67135	0.70743	0.71281
23517^-	0.49962	0.50259	<u>0.51939</u>	<u>0.51926</u>	**0.52082**	0.51941	<u>0.52033</u>	0.50635
40536	0.72550	0.85195	<u>0.86225</u>	<u>0.86291</u>	<u>0.86392</u>	<u>0.86128</u>	**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	<u>0.99978</u>	<u>0.99974</u>	0.99955	0.99253	**0.99987**
40701^-	0.76320	0.94313	0.95313	<u>0.95620</u>	**0.96000**	0.95007	<u>0.94533</u>	0.95370
40923^+	0.02127	0.78048	0.02169	0.74009	–	**0.89470**	0.86438	0.58220
40927^-	0.10096	0.35102	–	–	0.29429	<u>0.32001</u>	<u>0.32093</u>	**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	<u>0.99191</u>
40978^-	0.75346	<u>0.97368</u>	0.97114	**0.97774**	<u>0.97398</u>	0.96900	0.97358	–
40979^-	0.09983	0.95217	0.97367	<u>0.97783</u>	0.96883	<u>0.97750</u>	**0.98121**	0.97600
40981^-	0.49324	0.85604	0.85556	<u>0.87053</u>	0.86184	**0.89050**	<u>0.86913</u>	0.87633
40982^-	0.21681	0.74425	0.76364	<u>0.78268</u>	**0.79091**	0.76415	0.75955	<u>0.78062</u>
40983^-	0.89683	0.97886	<u>0.98581</u>	<u>0.98612</u>	<u>0.98540</u>	**0.98657**	0.95289	<u>0.98574</u>
40984^+	0.14473	0.93001	<u>0.93333</u>	0.93088	<u>0.94055</u>	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	<u>0.87341</u>
41027^+	0.42598	0.78945	0.85378	0.86775	<u>0.88735</u>	0.87540	<u>0.88691</u>	**0.90047**
41138	0.96474	0.99268	0.99137	0.99287	<u>0.99339</u>	0.97097	<u>0.99360</u>	**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	<u>0.82009</u>	**0.82366**	0.79911	0.80078	<u>0.80906</u>
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	<u>0.74347</u>	**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	<u>0.67841</u>
41167^-	0.00286	**0.87164**	–	–	–	0.38666	0.77971	–
41168^+	0.36200	0.65848	0.69273	**0.71814**	0.69642	0.63788	0.68494	<u>0.71786</u>
41169^+	0.02272	0.29082	<u>0.29566</u>	0.30692	**0.33576**	<u>0.32108</u>	0.28741	–
Average	0.44921	0.79980	0.80853	<u>0.82606</u>	**0.83040**	<u>0.80292</u>	0.81075	<u>0.82910</u>

Table 16: 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. Results not significantly worse than the best result—according to a Wilcoxon signed-rank test—are underlined. On data sets marked by $^+$ and $^-$, AutoML frameworks performed better and worse, respectively, than CASH solvers.
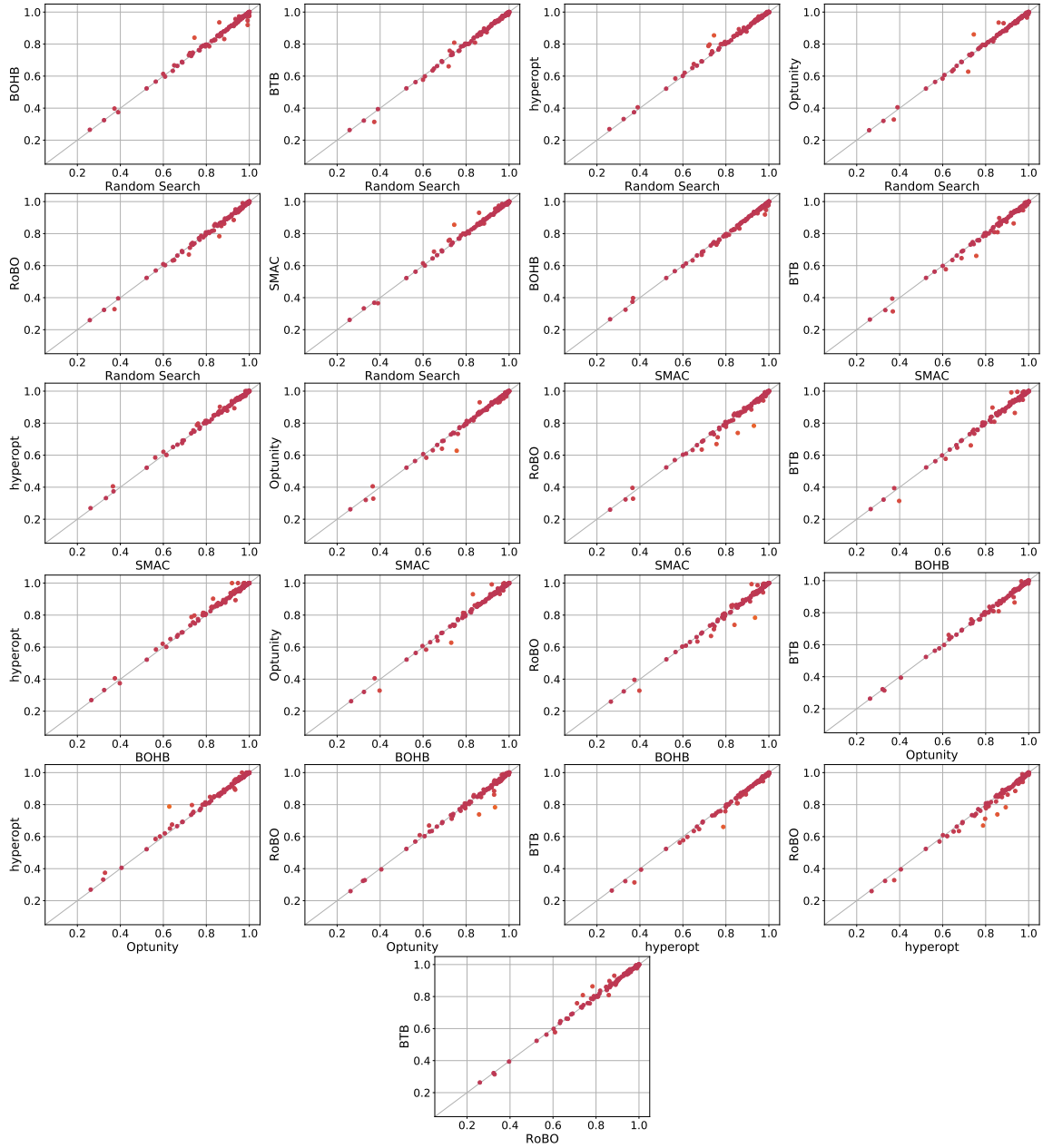
Figure 19: Pair-wise comparison of the mean precision of CASH algorithms. The axes represent the accuracy score of the stated CASH algorithm. Each point represents the averaged results for a single data set. Identical performances are plotted directly on the angle bisector. The comparison with grid search is omitted due to spacial constrictions.
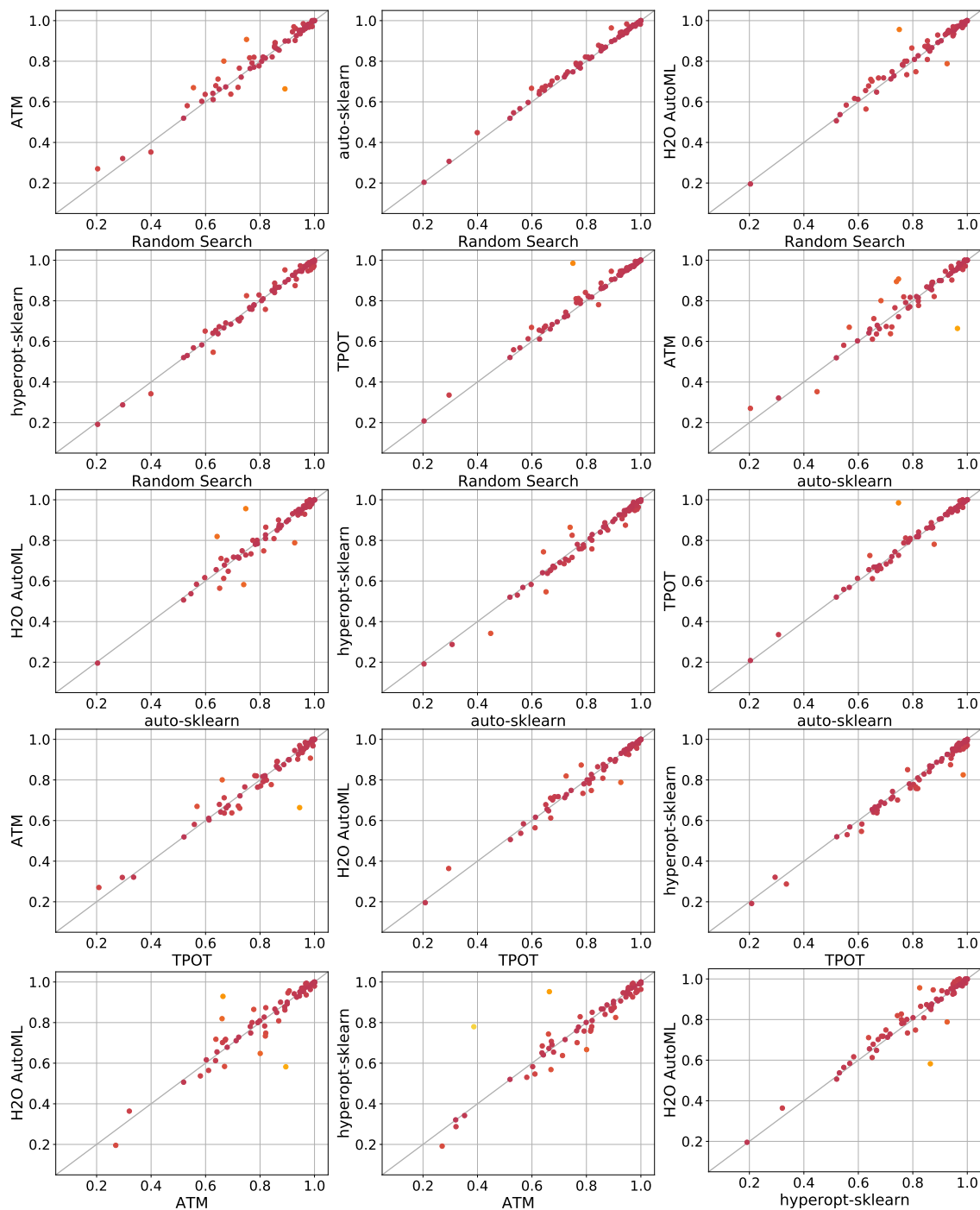
Figure 20: Pair-wise comparison of normalized performances of AutoML frameworks. The axes represent the accuracy score of the stated AutoML framework. Each point represents the averaged results for a single data set. Identical performances are plotted directly on the angle bisector.

References

Alaa, A. M., & Van Der Schaar, M. (2018). AutoPrognosis: Automated Clinical Prognostic Modeling via Bayesian Optimization with Structured Kernel Learning. *International Conference on Machine Learning*, *1*, 139–148.

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

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

Ayria, P. (2018). A complete Machine Learning PipeLine.. Available at `https://www.kaggle.com/pouryaayria/a-complete-ml-pipeline-tutorial-acu-86`.

Baidu (2018). EZDL.. Available at `http://ai.baidu.com/ezdl/`.

Balaji, A., & Allen, A. (2018). Benchmarking Automatic Machine Learning Frameworks. *arXiv preprint arXiv:1808.06492*.

Banzhaf, W., Nordin, P., Keller, R. E., & Francone, F. D. (1997). *Genetic Programming: An Introduction*. Morgan Kaufmann.

Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., & Mahajan, A. (2013). Mixed-integer nonlinear optimization. *Acta Numerica*, *22*, 1–131.

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798–1828.

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for Hyper-Parameter Optimization. In *International Conference on Neural Information Processing Systems*, pp. 2546–2554.

Bergstra, J., & Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, *13*, 281–305.

Bergstra, J., Yamins, D., & Cox, D. D. (2013). Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Python in Science Conference*, pp. 13–20.

Bilalli, B., Abelló, A., & Aluja-Banet, T. (2017). On the Predictive Power of Meta-Features in OpenML. *International Journal of Applied Mathematics and Computer Science*, *27*(4), 697–712.

Bischl, B., Casalicchio, G., Feurer, M., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N., & Vanschoren, J. (2017). OpenML Benchmarking Suites and the OpenML100. *arXiv preprint arXiv:1708.03731v1*.

Bischl, B., Casalicchio, G., Feurer, M., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N., & Vanschoren, J. (2019). OpenML Benchmarking Suites. *arXiv preprint arXiv:1708.03731v2*. arXiv:1708.03731.

Bottou, L. (2012). Stochastic Gradient Descent Tricks. In *Neural Networks, Tricks of the Trade, Reloaded*, pp. 430–445. Springer.

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

Breiman, L., Friedman, J., Stone, C. J., & Olsen, R. (1984). *Classification and Regression Trees*. Chapman and Hall.

Brochu, E., Cora, V. M., & de Freitas, N. (2010). A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1012.2599*.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Member, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games, 4*(1), 1–49.

Buyya, R. (1999). *High Performance Cluster Computing: Architectures and Systems*, Vol. 1. Prentice Hall.

Chan, T. (2017). Advisor.. Available at `https://github.com/tobegit3hub/advisor`.

Chapelle, O., Vapnik, V., Bousquet, O., & Mukherjee, S. (2002). Choosing Multiple Parameters for Support Vector Machines. *Machine Learning, 46*, 131–159.

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

Chen, P.-W., Wang, J.-Y., & Lee, H.-M. (2004). Model selection of SVMs using GA approach. In *IEEE International Joint Conference on Neural Networks*.

Chu, X., Ilyas, I. F., Krishnan, S., & Wang, J. (2016). Data Cleaning: Overview and Emerging Challenges. In *International Conference on Management of Data*, pp. 2201–2206.

Chu, X., Morcos, J., Ilyas, I. F., Ouzzani, M., Papotti, P., Tang, N., & Ye, Y. (2015). KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *ACM International Conference on Management of Data*, pp. 1247–1261.

Claesen, M., Simm, J., Popovic, D., Moreau, Y., & De Moor, B. (2014). Easy Hyperparameter Search Using Optunity. *arXiv preprint arXiv: 1412.1114*.

Clouder, A. (2018). Shortening Machine Learning Development Cycle with AutoML.. Available at `https://www.alibabacloud.com/blog/shortening-machine-learning-development-cycle-with-automl_594232`.

Coello, C. A. C., Lamont, G. B., & Van Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*, Vol. 5. Springer.

Das, P., Ivkin, N., Bansal, T., Rouesnel, L., Gautier, P., Karnin, Z., Dirac, L., Ramakrishnan, L., Perunicic, A., Shcherbatyi, I., Wu, W., Zolic, A., Shen, H., Ahmed, A., Winkelmolen, F., Miladinovic, M., Archembeau, C., Tang, A., Dutt, B., Grao, P., & Venkateswar, K. (2020). Amazon SageMaker Autopilot: a white box AutoML solution at scale Piali. In *Data Management for End-to-End Machine Learning*, pp. 1–7.

das Dôres, S. C. N., Soares, C., & Ruiz, D. (2018). Bandit-Based Automated Machine Learning. In *Brazilian Conference on Intelligent Systems*.

Dash, M., & Liu, H. (1997). Feature Selection for Classification. *Intelligent Data Analysis*, *1*, 131–156.

De Miranda, P. B., Prudêncio, R. B., De Carvalho, A. C. P., & Soares, C. (2012). 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*, pp. 562–575.

de Sá, A. G. C., Pinto, W. J. G. S., Oliveira, L. O. V. B., & Pappa, G. L. (2017). RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines. In *European Conference on Genetic Programming*, Vol. 10196, pp. 246–261.

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

Desautels, T., Krause, A., & Burdick, J. W. (2014). Parallelizing Exploration-Exploitation Tradeoffs with Gaussian Process Bandit Optimization. *Journal of Machine Learning Research*, *15*, 4053–4103.

Dinsmore, T. (2016). Automated Machine Learning: A Short History.. Available at `https://blog.datarobot.com/automated-machine-learning-short-history`.

Domhan, T., Springenberg, J. T., & Hutter, F. (2015). Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. *International Joint Conference on Artificial Intelligence*, pp. 3460–3468.

Dor, O., & Reich, Y. (2012). Strengthening learning algorithms by feature discovery. *Information Sciences*, *189*, 176–190.

Doshi-Velez, F., & Kim, B. (2017). Towards A Rigorous Science of Interpretable Machine Learning. *arXiv preprint arXiv:1702.08608*.

Drori, I., Krishnamurthy, Y., de Paula Lourenco, R., Rampin, R., Kyunghyun, C., Silva, C., & Freire, J. (2019). Automatic Machine Learning by Pipeline Synthesis using Model-Based Reinforcement Learning and a Grammar. In *International Conference on Machine Learning AutoML Workshop*.

Drori, I., Krishnamurthy, Y., Rampin, R., Lourenco, R. d. P., Ono, J. P., Cho, K., Silva, C., & Freire, J. (2018). AlphaD3M : Machine Learning Pipeline Synthesis. In *International Conference on Machine Learning AutoML Workshop*.

Eduardo, S., & Sutton, C. (2016). Data Cleaning using Probabilistic Models of Integrity Constraints. In *Neural Information Processing Systems*.

Efimova, V., Filchenkov, A., & Shalamov, V. (2017). Fast Automated Selection of Learning Algorithm And its Hyperparameters by Reinforcement Learning. In *International Conference on Machine Learning AutoML Workshop*.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., & Leyton-Brown, K. (2013). Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters. In *NIPS Workshop on Bayesian Optimization in Theory and Practice*.

Eggensperger, K., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2015). Efficient Benchmarking of Hyperparameter Optimizers via Surrogates. In *AAAI Conference on Artificial Intelligence*, pp. 1114–1120.

Eggensperger, K., Lindauer, M. T., Hoos, H. H., Hutter, F., & Leyton-Brown, K. (2018). Efficient Benchmarking of Algorithm Configuration Procedures via Model-Based Surrogates. *Machine Learning*, *107*, 15–41.

Elshawi, R., Maher, M., & Sakr, S. (2019). Automated Machine Learning: State-of-The-Art and Open Challenges. *arXiv preprint arXiv:1906.02287*.

Escalante, H. J., Montes, M., & Luis, V. (2009). Particle Swarm Model Selection for Authorship Verificatio. *Iberoamerican Congress on Pattern Recognition*, pp. 563–570.

Fabris, F., & Freitas, A. A. (2019). Analysing the Overfit of the auto-sklearn Automated Machine Learning Tool. In *Machine Learning, Optimization, and Data Science*, Vol. 11943, pp. 508–520. Springer International Publishing.

Falkner, S., Klein, A., & Hutter, F. (2018). BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *International Conference on Machine Learning*, pp. 1437–1446.

Fernández-Godino, M. G., Park, C., Kim, N.-H., & Haftka, R. T. (2016). Review of multi-fidelity models. *arXiv preprint arXiv:1609.07196*.

Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., & Hutter, F. (2018). Practical Automated Machine Learning for the AutoML Challenge 2018. *International Conference on Machine Learning AutoML Workshop*.

Feurer, M., & Hutter, F. (2018). Towards Further Automation in AutoML. In *International Conference on Machine Learning AutoML Workshop*.

Feurer, M., Klein, A., Eggensperger, K., Springenber, J. T., Blum, M., & Hutter, F. (2015a). Efficient and Robust Automated Machine Learning. In *International Conference on Neural Information Processing Systems*, pp. 2755–2763.

Feurer, M., Springenberg, J. T., & Hutter, F. (2015b). Initializing Bayesian Hyperparameter Optimization via Meta-Learning. *National Conference on Artificial Intelligence*, pp. 1128–1135.

Frazier, P. I. (2018). A Tutorial on Bayesian Optimization. *arXiv preprint arXiv: 1807.02811*, pp. 1–22.

Friedman, L., & Markovitch, S. (2015). Recursive Feature Generation for Knowledge-based Learning. *Journal of Artificial Intelligence Research*, *1*, 3–17.

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

Galhardas, H., Florescu, D., Shasha, D., & Simon, E. (2000). AJAX:An Extensible Data Cleaning Tool. In *International Conference on Management of Data*, pp. 590–596.

Gama, J., & Brazdil, P. (2000). Characterization of Classification Algorithms. In *Portuguese Conference on Artificial Intelligence*.

Garrido-Merchán, E. C., & Hernández-Lobato, D. (2018). Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes. In *International Conference on Machine Learning AutoML Workshop*, pp. 1–18.

Gaudel, R., & Sebag, M. (2010). Feature Selection as a One-Player Game. In *International Conference on Machine Learning*, pp. 359–366.

Geiger, A., Lenz, P., & Urtasun, R. (2012). Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition*.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planmning: Theory & Praxis*. Morgan Kaufmann Publishers, Inc.

Gijsbers, P., LeDell, E., Thomas, J., Poirier, S., Bischl, B., & Vanschoren, J. (2019). An Open Source AutoML Benchmark. In *International Conference on Machine Learning AutoML Workshop*.

Gil, Y., Honaker, J., Gupta, S., Ma, Y., Orazio, V. D., Garijo, D., Gadewar, S., Yang, Q., & Jahanshad, N. (2019). Towards Human-Guided Machine Learning. In *International Conference on Intelligent User Interfaces*.

Gil, Y., Yao, K.-T., Ratnakar, V., Garijo, D., Steeg, G. V., Szekely, P., Brekelmans, R., Kejriwal, M., Luo, F., & Huang, I.-H. (2018). P4ML: A Phased Performance-Based Pipeline Planner for Automated Machine Learning. In *International Conference on Machine Learning AutoML Workshop*, pp. 1–8.

Ginsbourger, D., Janusevskis, J., & Le Riche, R. (2010a). Dealing with asynchronicity in parallel Gaussian process based global optimization.. Available at `https://hal.archives-ouvertes.fr/hal-00507632`.

Ginsbourger, D., Le Riche, R., & Carraro, L. (2010b). Kriging Is Well-Suited to Parallelize Optimization. In *Computational Intelligence in Expensive Optimization Problems*, pp. 131–162. Springer Berlin Heidelberg.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., & Sculley, D. (2017). Google Vizier: A Service for Black-Box Optimization. In *ACM International Conference on Knowledge Discovery and Data Mining*, pp. 1487–1495.

Gomes, T. A., Prudêncio, R. B., Soares, C., Rossi, A. L., & Carvalho, A. (2012). Combining Meta-Learning and Search Techniques to Select Parameters for Support Vector Machines. *Neurocomputing*, *75*(1), 3–13.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Representation Learning. In *Deep Learning*, chap. 15. MIT Press.

Google LLC (2019). AI Explanations Whitepaper. Tech. rep., Google LLC.

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

Gustafson, L. (2018). *Bayesian Tuning and Bandits : An Extensible , Open Source Library for AutoML by*. Ph.D. thesis, Massachusetts Institute of Technology.

Guyon, I., Bennett, K., Cawley, G., Escalante, H. J., Escalera, S., Ho, T. K., Maciá, N., Ray, B., Saeed, M., Statnikov, A., & Viegas, E. (2015). Design of the 2015 ChaLearn AutoML Challenge. *International Joint Conference on Neural Networks*, pp. 1–8.

Guyon, I., Chaabane, I., Escalante, H. J., Escalera, S., Jajetic, D., Lloyd, J. R., Maciá, N., Ray, B., Romaszko, L., Sebag, M., Statnikov, A., Treguer, S., & Viegas, E. (2016). A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention. In *International Conference on Machine Learning AutoML Workshop*, pp. 21–30.

Guyon, I., & Elisseeff, A. (2003). An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, *3*, 1157–1182.

Guyon, I., Saffari, A., Dror, G., & Cawley, G. (2008). Analysis of the IJCNN 2007 Agnostic Learning vs. Prior Knowledge Challenge. *Neural Networks*, *21*(2-3), 544–550.

Guyon, I., Sun-Hosoya, L., Boullé, M., Escalante, H. J., Escalera, S., Liu, Z., Jajetic, D., Ray, B., Saeed, M., Sebag, M., Statnikov, A., Tu, W.-W., & Viegas, E. (2018). Analysis of the AutoML Challenge series 2015-2018. In *Automatic Machine Learning: Methods, Systems, Challenges*. Springer Verlag.

Guyon, I., Weston, J., & Barnhill, S. (2002). Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning*, *46*, 389–422.

H2O.ai (2018). H2O Driverless AI.. Available at `https://www.h2o.ai/products/h2o-driverless-ai/`.

H2O.ai (2019). H2O AutoML.. Available at `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`.

He, X., Zhao, K., & Chu, X. (2019). AutoML: A Survey of the State-of-the-Art. *arXiv preprint arXiv:1908.00709*.

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

Hennig, P., & Schuler, C. J. (2012). Entropy Search for Information-Efficient Global Optimization. *Journal of Machine Learning Research*, *13*, 1809–1837.

Hesterman, J. Y., Caucci, L., Kupinski, M. A., Barrett, H. H., & Furenlid, L. R. (2010). Maximum-Likelihood Estimation With a Contracting-Grid Search Algorithm. *IEEE Transactions on Nuclear Science*, *57*(3), 1077–1084.

Hoffman, M. W., Shahriari, B., & de Freitas, N. (2014). On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning. In *Artificial Intelligence and Statistics*, pp. 365–374.

Hsu, C.-W., Chang, C.-C., & Lin, C.-J. (2003). A Practical Guide to Support Vector Classification..

Huberman, B. A., Lukose, R. M., & Hogg, T. (1997). An Economics Approach to Hard Computational Problems. *Science*, *275*(5296), 51–54.

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

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2012). Parallel algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, Vol. 7219.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2014). An Efficient Approach for Assessing Hyperparameter Importance. In *International Conference on Machine Learning*, pp. 754–762.

Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research, 36*, 267–306.

Hutter, F., Kotthoff, L., & Vanschoren, J. (2018a). *Automated Machine Learning: Methods, Systems, Challenges.* Springer.

Hutter, F., Kotthoff, L., & Vanschoren, J. (2018b). Hyperparameter Optimization. In *Automatic Machine Learning: Methods, Systems, Challenges*, pp. 3–38. Springer.

Jamieson, K., & Talwalkar, A. (2015). Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Artificial Intelligence and Statistics*, pp. 240–248.

Jeffery, S. R., Alonso, G., Franklin, M. J., Hong, W., & Widom, J. (2006). Declarative Support for Sensor Data Cleaning. In *International Conference on Pervasive Computing*, pp. 83–100.

Kandasamy, K., Krishnamurthy, A., Schneider, J., & Póczos, B. (2018). Parallelised Bayesian Optimisation via Thompson Sampling Kirthevasan. In *International Conference on Artificial Intelligence and Statistics*, pp. 133–142.

Kanter, J. M., & Veeramachaneni, K. (2015). Deep Feature Synthesis: Towards Automating Data Science Endeavors. In *IEEE International Conference on Data Science and Advanced Analytics*, pp. 1–10.

Katz, G., Shin, E. C. R., & Song, D. (2017). ExploreKit: Automatic feature generation and selection. In *IEEE International Conference on Data Mining*, pp. 979–984.

Kaul, A., Maheshwary, S., & Pudi, V. (2017). AutoLearn - Automated Feature Generation and Selection. In *IEEE International Conference on Data Mining*.

Kégl, B. (2017). How to Build a Data Science Pipeline.. Available at `https://www.kdnuggets.com/2017/07/build-data-science-pipeline.html`.

Kennedy, J., & Eberhart, R. (1995). Particle Swarm Optimization. In *International Conference on Neural Networks*, pp. 1942–1948.

Khayyaty, Z., Ilyasz, I. F., Jindal, A., Madden, S., Ouzzani, M., Papotti, P., Quiané-Ruiz, J. A., Tang, N., & Yin, S. (2015). BigDansing: A System for Big Data Cleansing. In *ACM International Conference on Management of Data*, pp. 1215–1230.

Khurana, U., Samulowitz, H., & Turaga, D. (2018a). Ensembles with Automated Feature Engineering. In *International Conference on Machine Learning AutoML Workshop*.

Khurana, U., Samulowitz, H., & Turaga, D. (2018b). Feature Engineering for Predictive Modeling Using Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, pp. 3407–3414.

Khurana, U., Turaga, D., Samulowitz, H., & Parthasrathy, S. (2016). Cognito: Automated Feature Engineering for Supervised Learning. In *IEEE International Conference on Data Mining*, pp. 1304–1307.

Klein, A., Falkner, S., Bartels, S., Hennig, P., & Hutter, F. (2016). Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In *Artificial Intelligence and Statistics*, pp. 528–536.

Klein, A., Falkner, S., Mansur, N., & Hutter, F. (2017a). RoBO: A Flexible and Robust Bayesian Optimization Framework in Python. In *NIPS Bayesian Optimization Workshop*.

Klein, A., Falkner, S., Springenberg, J. T., & Hutter, F. (2017b). Learning Curve Prediction With Bayesian Neural Networks. *International Conference on Learning Representations*, pp. 1–16.

Koch, P., Golovidov, O., Gardner, S., Wujek, B., Griffin, J., & Xu, Y. (2018). Autotune: A Derivative-free Optimization Framework for Hyperparameter Tuning. In *ACM International Conference on Knowledge Discovery and Data Mining*, pp. 443–452.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning*, pp. 282–293.

Kohavi, R., & John, G. H. (1995). Automatic Parameter Selection by Minimizing Estimated Error. In *International Conference on Machine Learning*, pp. 304–312.

Komer, B., Bergstra, J., & Eliasmith, C. (2014). Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. In *International Conference on Machine Learning AutoML Workshop*, pp. 2825–2830.

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

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., & Leyton-Brown, K. (2016). Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *Journal of Machine Learning Research*, *17*, 1–5.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Krishnan, S., Wang, J., Franklin, M. J., Goldberg, K., Kraska, T., Milo, T., & Wu, E. (2015). SampleClean: Fast and Reliable Analytics on Dirty Data. *IEEE Data Engineering Bulletin*, *38*(3), 59–75.

Krishnan, S., Wang, J., Wu, E., Franklin, M. J., & Goldberg, K. (2016). ActiveClean: Interactive Data Cleaning For Statistical Modeling. In *Proceedings of the VLDB Endowment*, Vol. 12, pp. 948–959.

Krishnan, S., & Wu, E. (2019). AlphaClean: Automatic Generation of Data Cleaning Pipelines. *arXiv preprint arXiv:1904.11827*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *International Conference on Neural Information Processing Systems*, Vol. 1, pp. 1097–1105.

Lacoste, A., Larochelle, H., Marchand, M., & Laviolette, F. (2014). Sequential Model-Based Ensemble Optimization. In *Uncertainty In Artificial Intelligence*, pp. 440–448.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2017). Building machines that learn and think like people. *Behavioral and Brain Sciences*, *40*, 1–58.

Lam, H. T., Thiebaut, J.-M., Sinn, M., Chen, B., Mai, T., & Alkan, O. (2017). One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327*.

Langevin, S., Jonker, D., Bethune, C., Coppersmith, G., Hilland, C., Morgan, J., Azunre, P., & Gawrilow, J. (2018). Distil: A Mixed-Initiative Model Discovery System for Subject Matter Experts. In *International Conference on Machine Learning AutoML Workshop*.

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

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

Levesque, J. C., Durand, A., Gagne, C., & Sabourin, R. (2017). Bayesian Optimization for Conditional Hyperparameter Spaces. In *International Joint Conference on Neural Networks*, pp. 286–293.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Hardt, M., Recht, B., & Talwalkar, A. (2020). A System for Massively Parallel Hyperparameter Tuning. In *Machine Learning and Systems*.

Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2016). Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *arXiv preprint arXiv:1603.06560*.

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

Lindauer, M., & Hutter, F. (2018). Warmstarting of Model-based Algorithm Configuration. In *AAAI Conference on Artificial Intelligence*, pp. 1355–1362.

Luo, G. (2016). 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.

Maclaurin, D., Duvenaud, D., & Adams, R. P. (2015). Gradient-based Hyperparameter Optimization through Reversible Learning. In *International Conference on Machine Learning*, pp. 2113–2122.

Margaritis, D. (2009). Toward Provably Correct Feature Selection in Arbitrary Domains. In *Neural Information Processing Systems*, pp. 1240–1248.

Markovitch, S., & Rosenstein, D. (2002). Feature generation using general constructor functions. *Machine Learning*, *49*(1), 59–98.

Maron, O., & Moore, A. (1993). Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. *Advances in Neural Information Processing Systems*, pp. 59–66.

McGushion, H. (2019). HyperparameterHunter.. Available at `https://github.com/HunterMcGushion/hyperparameter_hunter`.

Meinshausen, N., & Bühlmann, P. (2010). Stability selection. *Journal of the Royal Statistical Society*, *72*(4), 417–473.

Mejía-Lavalle, M., Sucar, E., & Arroyo, G. (2006). Feature Selection With A Perceptron Neural Net. In *International Workshop on Feature Selection for Data Mining*, pp. 131–135.

Messaoud, I. B., El Abed, H., Märgner, V., & Amiri, H. (2011). A design of a preprocessing framework for large database of historical documents. In *Workshop on Historical Document Imaging and Processing*, pp. 177–183.

Mohr, F., Wever, M., & Hüllermeier, E. (2018). ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, *107*, 1495–1515.

Momma, M., & Bennett, K. P. (2002). A Pattern Search Method for Model Selection of Support Vector Regression. In *SIAM International Conference on Data Mining*, pp. 261–274.

Motoda, H., & Liu, H. (2002). Feature Selection, Extraction and Construction. *Communication of Institute of Information and Computing Machinery*, *5*, 67–72.

Munos, R. (2014). From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning. Tech. rep., hal-00747575.

Nargesian, F., Samulowitz, H., Khurana, U., Khalil, E. B., & Turaga, D. (2017). Learning Feature Engineering for Classification. In *International Joint Conference on Artificial Intelligence*, pp. 2529–2535.

Nguyen, T.-D., Maszczyk, T., Musial, K., Zöller, M.-A., & Gabrys, B. (2020). AVATAR - Machine Learning Pipeline Evaluation Using Surrogate Model. In *International Symposium on Intelligent Data Analysis*, pp. 352–365.

Nickson, T., Osborne, M. A., Reece, S., & Roberts, S. (2014). Automated Machine Learning on Big Data using Stochastic Algorithm Tuning. *arXiv preprint arXiv: 1407.7969*.

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

Olson, R. S., & Moore, J. H. (2016). TPOT : A Tree-based Pipeline Optimization Tool for Automating Machine Learning. In *International Conference on Machine Learning AutoML Workshop*, pp. 66–74.

Olson, R. S., Urbanowicz, R. J., Andrews, P. C., Lavender, N. A., Kidd, L. C., & Moore, J. H. (2016). Automating biomedical data science through tree-based pipeline optimization. In *Applications of Evolutionary Computation*, pp. 123–137. Springer International Publishing.

Opitz, D., & Maclin, R. (1999). Popular Ensemble Methods: An Empirical Study. *Journal of Artificial Intelligence Research*, *11*, 169–198.

Parry, P. (2019). auto_ml.. Available at `https://github.com/ClimbsRocks/auto_ml`.

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

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pedregosa, F. (2016). Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, pp. 737–746.

Perrone, V., Shen, H., Seeger, M., Archambeau, C., & Jenatton, R. (2019). Learning search spaces for Bayesian optimization: Another view of hyperparameter transfer learning. In *Advances in Neural Information Processing Systems 32*, pp. 12771—-12781. Curran Associates, Inc.

Petri, C. A. (1962). *Kommunikation mit Automaten*. Ph.D. thesis, Universität Hamburg.

Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A Field Guide to Genetic Programing*. Lulu.com.

Polikar, R. (2006). Ensemble Based Systems in Decision Making. *IEEE Circuits and Systems Magazine*, *6*(3), 21–45.

Post, M. J., van der Putten, P., & van Rijn, J. N. (2016). Does Feature Selection Improve Classification? A Large Scale Experiment in OpenML. In *Advances in Intelligent Data Analysis XV*, pp. 158–170.

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

Probst, P., Boulesteix, A.-L., & Bischl, B. (2019). Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *Journal of Machine Learning Research*, *20*(53), 1–32.

Pudil, P., Novovičová, J., & Kittler, J. (1994). Floating search methods in feature selection. *Pattern recognition letters*, *15*(11), 1119–1125.

Pyle, D. (1999). *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, Inc.

Quanming, Y., Mengshuo, W., Hugo, J. E., Isabelle, G., Yi-Qi, H., Yu-Feng, L., Wei-Wei, T., Qiang, Y., & Yang, Y. (2018). Taking Human out of Learning Applications: A Survey on Automated Machine Learning. *arXiv preprint arXiv:1810.13306*.

Rahm, E., & Do, H. H. (2000). Data cleaning: Problems and Current Approaches. In *IEEE Data Engineering Bulletin*.

Rakotoarison, H., Schoenauer, M., & Sebag, M. (2019). Automated Machine Learning with Monte-Carlo Tree Search. In *International Joint Conference on Artificial Intelligence*, pp. 3296–3303.

Rakotomamonjy, A. (2003). Variable selection using SVM-based criteria. *Journal of Machine Learning Research*, *3*, 1357–1370.

Raman, V., & Hellerstein, J. M. (2001). Potter's Wheel: An Interactive Data Cleaning System. In *International Conference on Very Large Data Bases*, Vol. 1, pp. 381–390.

RapidMiner (2018). Introducing RapidMiner Auto Model.. Available at `https://rapidminer.com/resource/automated-machine-learning/`.

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

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

Reif, M., Shafait, F., & Dengel, A. (2012). Meta-learning for evolutionary parameter optimization of classifier. *Machine Learning*, *87*, 357–380.

Rekatsinas, T., Chuy, X., Ilyasy, I. F., & Ré, C. (2017). HoloClean: Holistic Data Repairs with Probabilistic Inference. In *VLDB Endowment*, pp. 1190–1201.

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

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

Rokach, L. (2010). Ensemble-based classifiers. *Artificial Intelligence Review*, *33*(1-2), 1–39.

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

Saeys, Y., Inza, I., & Larrañaga, P. (2007). A review of feature selection techniques in bioinformatics. *Bioinformatics*, *23*(19), 2507–2517.

Salvador, M. M., Budka, M., & Gabrys, B. (2016). Towards automatic composition of multicomponent predictive systems. In *International Conference on Hybrid Artificial Intelligence Systems*, pp. 27–39.

Salvador, M. M., Budka, M., & Gabrys, B. (2017). Modelling multi-component predictive systems as petri nets. In *Industrial Simulation Conference*, pp. 17–23.

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

Schoenfeld, B., Giraud-Carrier, C., Poggemann, M., Christensen, J., & Seppi, K. (2018). Preprocessor Selection for Machine Learning Pipelines. In *International Conference on Machine Learning AutoML Workshop*.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & de Freitas, N. (2016). Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*, *104*(1), 148 – 175.

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

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*.

Smith, M. G., & Bull, L. (2005). Genetic Programming with a Genetic Algorithm for Feature Construction and Selection. *Genetic Programming and Evolvable Machines*, *6*(3), 265–281.

Smith, M. J., Wedge, R., & Veeramachaneni, K. (2017). FeatureHub: Towards collaborative data science. In *IEEE International Conference on Data Science and Advanced Analytics*, pp. 590–600.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, pp. 2951–2959.

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

Sohn, S. Y. (1999). Meta analysis of classification algorithms for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *21*(11), 1137–1144.

Solis, F. J., & Wets, R. J.-B. (1981). Minimization By Random Search Techniques. *Mathematics of Operations Research*, *6*(1), 19–30.

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

Sparks, E. R., Talwalkar, A., Haas, D., Franklin, M. J., Jordan, M. I., & Kraska, T. (2015). Automating model search for large scale machine learning. In *ACM Symposium on Cloud Computing*, pp. 368–380.

Swearingen, T., Drevo, W., Cyphers, B., Cuesta-Infante, A., Ross, A., & Veeramachaneni, K. (2017). ATM: A distributed, collaborative, scalable system for automated machine learning. In *IEEE International Conference on Big Data*, pp. 151–162.

Swersky, K., Snoek, J., & Adams, R. P. (2014). Freeze-Thaw Bayesian Optimization. *arXiv preprint arXiv:1406.3896*.

Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2013). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *ACM International Conference on Knowledge Discovery and Data Mining*, pp. 847–855.

Tran, B., Xue, B., & Zhang, M. (2016). Genetic programming for feature construction and selection in classification on high-dimensional data. *Memetic Computing*, *8*, 3–15.

Tuggener, L., Amirian, M., Rombach, K., Lörwald, S., Varlet, A., Westermann, C., & Stadelmann, T. (2019). Automated Machine Learning in Practice: State of the Art and Recent Results. In *Swiss Conference on Data Science*, pp. 31–36.

Tuv, E., Borisov, A., Runger, G., & Torkkola, K. (2009). Feature Selection with Ensembles, Artificial Variables, and Redundancy Elimination. *Journal of Machine Learning Research*, *10*, 1341–1366.

USU Software AG (2018). Katana.. Available at `https://katana.usu.de/`.

Vafaie, H., & De Jong, K. (1992). Genetic Algorithms as a Tool for Feature Selection in Machine Learning. In *International Conference on Tools with Artificial Intelligence*, pp. 200–203.

van Rijn, J. N., Abdulrahman, S. M., Brazdil, P., & Vanschoren, J. (2015). Fast Algorithm Selection Using Learning Curves. In *International Symposium on Intelligent Data Analysis*.

van Rijn, J. N., & Hutter, F. (2018). Hyperparameter Importance Across Datasets. In *International Conference on Knowledge Discovery and Data Mining*, pp. 2367–2376.

Vanschoren, J. (2019). Meta-Learning. In *Automatic Machine Learning: Methods, Systems, Challenges*, pp. 35–61. Springer.

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

Weisz, G., Gyorgy, A., & Szepesvari, C. (2018). LeapsAndBounds: A Method for Approximately Optimal Algorithm Configuration. In *International Conference on Machine Learning AutoML Workshop*, pp. 5257–5265.

Wever, M., Mohr, F., & Hüllermeier, E. (2018). ML-Plan for Unlimited-Length Machine Learning Pipelines. In *International Conference on Machine Learning AutoML Workshop*.

Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2015a). 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*, pp. 104–119.

Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2015b). Learning Hyperparameter Optimization Initializations. In *IEEE International Conference on Data Science and Advanced Analytics*.

Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2017). Automatic Frankensteining: Creating Complex Ensembles Autonomously. In *SIAM International Conference on Data Mining*, pp. 741–749.

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

Yang, Y., & Pedersen, J. O. (1997). A Comparative Study on Feature Selection in Text Categorization. *International Conference on Machine Learning*, *97*, 412–420.

Zhang, Y., Bahadori, M. T., Su, H., & Sun, J. (2016). FLASH: Fast Bayesian Optimization for Data Analytic Pipelines. In *ACM International Conference on Knowledge Discovery and Data Mining*, pp. 2065–2074.

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