Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms

Chris Thornton Frank Hutter Holger H. Hoos Kevin Leyton-Brown

Department of Computer Science, University of British Columbia
201-2366 Main Mall, Vancouver BC, V6T 1Z4, Canada
{cwthornt, hutter, hoos, kevinlb}@cs.ubc.ca

ABSTRACT

Many different machine learning algorithms exist; taking into account each algorithm's hyperparameters, there is a staggeringly large number of possible alternatives overall. We consider the problem of simultaneously selecting a learning algorithm and setting its hyperparameters, going beyond previous work that attacks these issues separately. We show that this problem can be addressed by a fully automated approach, leveraging recent innovations in Bayesian optimization. Specifically, we consider a wide range of feature selection techniques (combining 3 search and 8 evaluator methods) and all classification approaches implemented in WEKA's standard distribution, spanning 2 ensemble methods, 10 meta-methods, 27 base classifiers, and hyperparameter settings for each classifier. On each of 21 popular datasets from the UCI repository, the KDD Cup 09, variants of the MNIST dataset and CIFAR-10, we show classification performance often much better than using standard selection and hyperparameter optimization methods. We hope that our approach will help non-expert users to more effectively identify machine learning algorithms and hyperparameter settings appropriate to their applications, and hence to achieve improved performance.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.2.2 [**Artificial Intelligence**]: Automatic Programming; G.1.6 [**Mathematics of Computing**]: Optimization

General Terms

Algorithms, Performance, Experimentation

Keywords

Model selection; Hyperparameter optimization; WEKA

1. INTRODUCTION

Increasingly, users of machine learning tools are non-experts who require off-the-shelf solutions. The machine learning community has much aided such users by making available a wide variety of sophisticated learning algorithms and feature selection methods through open source packages, such as WEKA [14] and PyBrain [26]. Such packages ask a user to make two kinds of choices: selecting a learning algorithm and customizing it by setting hyperparameters (which also control feature selection, if applicable). It can be challenging to make the right choice when faced with these degrees of freedom, leaving many users to select algorithms based on reputation or intuitive appeal, and/or to leave hyperparameters set to default values. Of course, adopting this approach can yield performance far worse than that of the best method and hyperparameter settings.

This suggests a natural challenge for machine learning: given a dataset, automatically and simultaneously choosing a learning algorithm and setting its hyperparameters to optimize empirical performance. We dub this the *combined algorithm selection and hyperparameter optimization (CASH) problem*; we formally define it in Section 3. There has been considerable past work separately addressing model selection [*e.g.*, 1, 6, 7, 8, 10, 23, 24, 34] and hyperparameter optimization [*e.g.*, 3, 4, 5, 13, 29, 31, 22]. In contrast, despite its practical importance, we are surprised to find only limited variants of the CASH problem in the literature; furthermore, these consider a fixed and relatively small number of parameter configurations for each algorithm [see, e.g., 21].

A likely explanation is that it is very challenging to search the combined space of learning algorithms and their hyperparameters: the response function is noisy and the space is high dimensional, involves both categorical and continuous choices, and contains hierarchical dependencies (*e.g.*, the hyperparameters of a learning algorithm are only meaningful if that algorithm is chosen; the algorithm choices in an ensemble method are only meaningful if that ensemble method is chosen; etc). Another related line of work is on meta-learning procedures that exploit characteristics of the dataset, such as the performance of so-called landmarking algorithms, to predict which algorithm or hyperparameter configuration will perform well [2, 21, 25, 32]. While the CASH algorithms we study in this paper start from scratch for each new dataset, these meta-learning procedures exploit information from previous datasets, which may not always be available.

In what follows, we demonstrate that CASH can be viewed as a single hierarchical hyperparameter optimization problem, in which even the choice of algorithm itself is considered a hyperparameter. We also show that—based on this problem formulation—recent Bayesian optimization methods can obtain high quality results in reasonable time and with minimal human effort. After discussing some preliminaries (Section

KDD 2013 Chicago, Illinois, USA
ACM 978-1-4503-2174-7/13/08 ...$15.00.

2), we define the CASH problem and discuss methods for tackling it (Section 3). We then define a concrete CASH problem encompassing the full range of classifiers and feature selectors in the open source package WEKA (Section 4), and show that a search in the combined space of algorithms and hyperparameters yields better-performing models than standard algorithm selection and hyperparameter optimization methods (Section 5). More specifically, we show that the recent Bayesian optimization procedures TPE [4] and SMAC [15] often find combinations of algorithms and hyperparameters that outperform existing baseline methods, especially on large datasets.

2. PRELIMINARIES

This work focuses on classification problems: learning a function $f : \mathcal{X} \mapsto \mathcal{Y}$ with finite $\mathcal{Y}$. A *learning algorithm* A maps a set $\{d_1, \ldots, d_n\}$ of training data points $d_i = (\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ to such a function, which is often expressed via a vector of *model parameters*. Most learning algorithms A further expose *hyperparameters* $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$, which change the way the learning algorithm $A_{\boldsymbol{\lambda}}$ itself works. For example, hyperparameters are used to describe a description-length penalty, the number of neurons in a hidden layer, the number of data points that a leaf in a decision tree must contain to be eligible for splitting, etc. These hyperparameters are typically optimized in an "outer loop" that evaluates the performance of each hyperparameter configuration using cross-validation.

2.1 Model Selection

Given a set of learning algorithms $\mathcal{A}$ and a limited amount of training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, the goal of model selection is to determine the algorithm $A^* \in \mathcal{A}$ with optimal generalization performance. Generalization performance is estimated by splitting $\mathcal{D}$ into disjoint training and validation sets $\mathcal{D}^{(i)}_{\text{train}}$ and $\mathcal{D}^{(i)}_{\text{valid}}$, learning functions f_i by applying A^* to $\mathcal{D}^{(i)}_{\text{train}}$, and evaluating the predictive performance of these functions on $\mathcal{D}^{(i)}_{\text{valid}}$. This allows for the model selection problem to be written as:

$$A^* \in \operatorname*{argmin}_{A \in \mathcal{A}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(A, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}),$$

where $\mathcal{L}(A, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}})$ is the loss (here: misclassification rate) achieved by A when trained on $\mathcal{D}^{(i)}_{\text{train}}$ and evaluated on $\mathcal{D}^{(i)}_{\text{valid}}$.

We use k-fold cross-validation [19], which splits the training data into k equal-sized partitions $\mathcal{D}^{(1)}_{\text{valid}}, \ldots, \mathcal{D}^{(k)}_{\text{valid}}$, and sets $\mathcal{D}^{(i)}_{\text{train}} = \mathcal{D} \setminus \mathcal{D}^{(i)}_{\text{valid}}$ for $i = 1, \ldots, k$.[1]

2.2 Hyperparameter Optimization

The problem of optimizing the hyperparameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ of a given learning algorithm A is conceptually similar to that of model selection. Some key differences are that hyperparameters are often continuous, that hyperparameter spaces are often high dimensional, and that we can exploit correlation structure between different hyperparameter settings $\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2 \in \boldsymbol{\Lambda}$. Given n hyperparameters $\lambda_1, \ldots, \lambda_n$ with domains $\Lambda_1, \ldots, \Lambda_n$, the hyperparameter space $\boldsymbol{\Lambda}$ is a subset of the crossproduct of these domains: $\boldsymbol{\Lambda} \subset \Lambda_1 \times \cdots \times \Lambda_n$.

[1]There are other ways of estimating generalization performance; e.g., we also experimented with repeated random subsampling validation [19], and obtained similar results.

Algorithm 1 SMBO

1: initialise model $\mathcal{M}_L$; $\mathcal{H} \leftarrow \emptyset$
2: **while** time budget for optimization has not been exhausted **do**
3: $\quad\boldsymbol{\lambda} \leftarrow$ candidate configuration from $\mathcal{M}_L$
4: $\quad$Compute $c = \mathcal{L}(A_{\boldsymbol{\lambda}}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}})$
5: $\quad\mathcal{H} \leftarrow \mathcal{H} \cup \{(\boldsymbol{\lambda}, c)\}$
6: $\quad$Update $\mathcal{M}_L$ given $\mathcal{H}$
7: **end while**
8: **return** $\boldsymbol{\lambda}$ from $\mathcal{H}$ with minimal c

This subset is often strict, such as when certain settings of one hyperparameter render other hyperparameters inactive. For example, the parameters determining the specifics of the third layer of a deep belief network are not relevant if the network depth is set to one or two. Likewise, the parameters of a support vector machine's polynomial kernel are not relevant if we use a different kernel instead.

More formally, following [16], we say that a hyperparameter λ_i is *conditional* on another hyperparameter λ_j, if λ_i is only active if hyperparameter λ_j takes values from a given set $V_i(j) \subsetneq \Lambda_j$; in this case we call λ_j a *parent* of λ_i. Conditional hyperparameters can in turn be parents of other conditional hyperparameters, giving rise to a tree-structured space [4] or, in some cases, a directed acyclic graph (DAG) [16]. Given such a structured space $\boldsymbol{\Lambda}$, the (hierarchical) hyperparameter optimization problem can be written as:

$$\boldsymbol{\lambda}^* \in \operatorname*{argmin}_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(A_{\boldsymbol{\lambda}}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}).$$

3. COMBINED ALGORITHM SELECTION AND HYPERPARAMETER OPTIMIZATION (CASH)

Given a set of algorithms $\mathcal{A} = \{A^{(1)}, \ldots, A^{(k)}\}$ with associated hyperparameter spaces $\boldsymbol{\Lambda}^{(1)}, \ldots, \boldsymbol{\Lambda}^{(k)}$, we define the combined algorithm selection and hyperparameter optimization problem (CASH) as computing

$$A^*_{\boldsymbol{\lambda}^*} \in \operatorname*{argmin}_{A^{(j)} \in \mathcal{A}, \boldsymbol{\lambda} \in \boldsymbol{\Lambda}^{(j)}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(A^{(j)}_{\boldsymbol{\lambda}}, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}). \quad (1)$$

We note that this problem can be reformulated as a single combined hierarchical hyperparameter optimization problem with parameter space $\boldsymbol{\Lambda} = \boldsymbol{\Lambda}^{(1)} \cup \cdots \cup \boldsymbol{\Lambda}^{(k)} \cup \{\lambda_r\}$, where λ_r is a new root-level hyperparameter that selects between algorithms $A^{(1)}, \ldots, A^{(k)}$. The root-level parameters of each subspace $\boldsymbol{\Lambda}^{(i)}$ are made conditional on λ_r being instantiated to A_i.

In principle, Problem 1 can be tackled in various ways. A promising approach is Bayesian Optimization [9], and in particular Sequential Model-Based Optimization [SMBO; 15], a versatile stochastic optimization framework that can work explicitly with both categorical and continuous hyperparameters, and that can exploit hierarchical structure stemming from conditional parameters. SMBO (outlined in Algorithm 1) first builds a model $\mathcal{M}_{\mathcal{L}}$ that captures the dependence of loss function $\mathcal{L}$ on hyperparameter settings $\boldsymbol{\lambda}$ (line 1 in Algorithm 1). It then iterates the following steps: use $\mathcal{M}_{\mathcal{L}}$ to determine a promising candidate configuration of hyperparameters $\boldsymbol{\lambda}$ to evaluate next (line 3); evaluate the loss c of $\boldsymbol{\lambda}$ (line 4); and update the model $\mathcal{M}_{\mathcal{L}}$ with the new data point $(\boldsymbol{\lambda}, c)$ thus obtained (lines 5–6).

In order to select its next hyperparameter configuration $\boldsymbol{\lambda}$ using model $\mathcal{M}_{\mathcal{L}}$, SMBO uses a so-called *acquisition func-*

tion $a_{\mathcal{M}_\mathcal{L}} : \mathbf{\Lambda} \mapsto \mathbb{R}$, which uses the predictive distribution of model $\mathcal{M}_\mathcal{L}$ at arbitrary hyperparameter configurations $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ to quantify (in closed form) how useful knowledge about $\boldsymbol{\lambda}$ would be. SMBO then simply maximizes this function over $\mathbf{\Lambda}$ to select the most useful configuration $\boldsymbol{\lambda}$ to evaluate next. Several well-studied acquisition functions exist [18, 27, 30]; all aim to automatically trade off exploitation (locally optimizing hyperparameters in regions known to perform well) versus exploration (trying hyperparameters in a relatively unexplored region of the space) in order to avoid premature convergence. In this work, we maximized *positive expected improvement (EI)* attainable over an existing given error rate c_{min} [27]. Let $c(\boldsymbol{\lambda})$ denote the error rate of hyperparameter configuration $\boldsymbol{\lambda}$. Then, the positive improvement function over c_{min} is defined as

$$I_{c_{min}}(\boldsymbol{\lambda}) := \max\{c_{min} - c(\boldsymbol{\lambda}), 0\}.$$

Of course, we do not know $c(\boldsymbol{\lambda})$. We can, however, compute its expectation with respect to the current model $\mathcal{M}_\mathcal{L}$:

$$\mathbb{E}_{\mathcal{M}_\mathcal{L}}[I_{c_{min}}(\boldsymbol{\lambda})] = \int_{-\infty}^{c_{min}} \max\{c_{min} - c, 0\} \cdot p_{\mathcal{M}_L}(c \mid \lambda)\, dc. \tag{2}$$

One main difference between existing SMBO algorithms lies in the model class they employ. We now review the two whose models can handle hierarchical hyperparameters and that are thus suitable for the CASH problem.

3.1 Sequential Model-based Algorithm Configuration (SMAC)

Sequential model-based algorithm configuration [SMAC; 15] supports a variety of models $p(c \mid \boldsymbol{\lambda})$ to capture the dependence of the loss function c on hyper-parameters $\boldsymbol{\lambda}$, including approximate Gaussian processes and random forests. In this paper we use random forest models, since they tend to perform well with discrete and high-dimensional input data. SMAC handles conditional parameters by instantiating inactive conditional parameters in $\boldsymbol{\lambda}$ to default values for model training and prediction. This allows the individual decision trees to include splits of the kind "is hyperparameter λ_i active?", allowing them to focus on active hyperparameters. While random forests are not usually treated as probabilistic models, SMAC obtains a predictive mean $\mu_{\boldsymbol{\lambda}}$ and variance ${\sigma_{\boldsymbol{\lambda}}}^2$ of $p(c \mid \boldsymbol{\lambda})$ as frequentist estimates over the predictions of its individual trees for $\boldsymbol{\lambda}$; it then models $p_{\mathcal{M}_\mathcal{L}}(c \mid \boldsymbol{\lambda})$ as a Gaussian $\mathcal{N}(\mu_{\boldsymbol{\lambda}}, {\sigma_{\boldsymbol{\lambda}}}^2)$.

SMAC uses the expected improvement criterion defined in Equation 2, instantiating c_{min} to the error rate of the best hyperparameter configuration measured so far. Under SMAC's predictive distribution $p_{\mathcal{M}_\mathcal{L}}(c \mid \boldsymbol{\lambda}) = \mathcal{N}(\mu_{\boldsymbol{\lambda}}, {\sigma_{\boldsymbol{\lambda}}}^2)$, this expectation is the closed-form expression

$$\mathbb{E}_{\mathcal{M}_\mathcal{L}}[I_{c_{min}}(\boldsymbol{\lambda})] = \sigma_{\boldsymbol{\lambda}} \cdot [u \cdot \Phi(u) + \varphi(u)],$$

where $u = \frac{c_{\min} - \mu_{\boldsymbol{\lambda}}}{\sigma_{\boldsymbol{\lambda}}}$, and φ and Φ denote the probability density function and cumulative distribution function of a standard normal distribution, respectively [18].

SMAC is designed for robust optimization under noisy function evaluations, and as such implements special mechanisms to keep track of its best known configuration and assure high confidence in its estimate of that configuration's performance. This robustness against noisy function evaluations can be exploited in combined algorithm selection and hyperparameter optimization, since the function to be optimized in Equation (1) is a mean over a set of loss terms (each corresponding to one pair of $\mathcal{D}^{(i)}_{\text{train}}$ and $\mathcal{D}^{(i)}_{\text{valid}}$ constructed from the training set). A key idea in SMAC is to make progressively better estimates of this mean by evaluating these terms one at a time, thus trading off accuracy and computational cost. In order for a new configuration to become a new incumbent, it must outperform the previous incumbent in every comparison made: considering only one fold, two folds, and so on up to the total number of folds previously used to evaluate the incumbent. (Furthermore, every time the incumbent survives such a comparison, it is evaluated on a new fold, up to the total number available, meaning that the number of folds used to evaluate the incumbent grows over time.) A poorly performing configuration can thus be discarded after considering just a single fold.

Finally, SMAC also implements a diversification mechanism to achieve robust performance even when its model is misled, and to explore new parts of the space: every second configuration is selected at random. Because of the evaluation procedure just described, this requires less overhead than one might imagine.

3.2 Tree-structured Parzen Estimator (TPE)

While SMAC models $p(c \mid \boldsymbol{\lambda})$ explicitly, the Tree-structure Parzen Estimator [TPE; 4] uses separate models for $p(c)$ and $p(\boldsymbol{\lambda} \mid c)$. Specifically, it models $p(\boldsymbol{\lambda} \mid c)$ as one of two density estimates, conditional on whether c is greater or less than a given threshold value c^*:

$$p(\boldsymbol{\lambda} \mid c) = \begin{cases} \ell(\boldsymbol{\lambda}), & \text{if } c < c^*. \\ g(\boldsymbol{\lambda}), & \text{if } c \geq c^*. \end{cases} \tag{3}$$

Here, c^* is chosen as the γ-quantile of the losses TPE obtained so far (where γ is an algorithm parameter with a default value of $\gamma = 0.15$), $\ell(\cdot)$ is a density estimate learned from all previous hyperparameters $\boldsymbol{\lambda}$ with corresponding loss smaller than c^*, and $g(\cdot)$ is a density estimate learned from all previous hyperparameters $\boldsymbol{\lambda}$ with corresponding loss greater than or equal to c^*. Intuitively, this creates a probabilistic density estimator $\ell(\cdot)$ for hyperparameters that appear to do 'well', and a different density estimator $g(\cdot)$ for hyperparameters that appear 'poor' with respect to the threshold. Bergstra et al. [4] showed that the expected improvement $\mathbb{E}_{\mathcal{M}_\mathcal{L}}[I_{c_{min}}(\boldsymbol{\lambda})]$ from Equation 2 is proportional to closed-form expression:

$$\mathbb{E}[I_{c_{min}}(\boldsymbol{\lambda})] \propto \left(\gamma + \frac{g(\boldsymbol{\lambda})}{\ell(\boldsymbol{\lambda})} \cdot (1 - \gamma)\right)^{-1}.$$

TPE maximizes this expression by generating many candidate hyperparameter configurations at random and picking a $\boldsymbol{\lambda}$ that minimizes $g(\boldsymbol{\lambda})/\ell(\boldsymbol{\lambda})$.

The density estimators $\ell(\cdot)$ and $g(\cdot)$ have a hierarchical structure with discrete, continuous, and conditional variables reflecting the hyperparameters and their dependence relationships. For each node in this tree structure, a 1-D Parzen estimator is created to model the density of the node's corresponding hyperparameter. For a given hyperparameter configuration $\boldsymbol{\lambda}$ that is added to either ℓ or g, only the 1-D estimators corresponding to active hyperparameters in $\boldsymbol{\lambda}$ are updated. For continuous hyperparameters, these 1-D estimators are constructed by placing density in the form of a Gaussian at each hyperparameter value $\boldsymbol{\lambda}_i$, with standard deviation set to the larger of each point's left and right neighbours. Discrete hyperparameters are estimated with probabilities proportional to the number of times that a particular choice occurred in the set of observations. To evaluate a candidate hyperparameter $\boldsymbol{\lambda}$'s probability estimate, TPE starts at the root of the tree and descends into the leaves

Table 1: Classifiers in Auto-WEKA. * indicates meta-methods, which in addition to their own parameters take one 'base' classifier and its parameters. + indicates ensemble methods that take as input up to 5 'base' classifiers and their parameters. We report the number of *Categorical* and *Numeric* hyperparameters for each method.

Classifier	Categorical	Numeric
Bayes Net	2	0
Naive Bayes	2	0
Naive Bayes Multinomial	0	0
Gaussian Process	3	6
Linear Regression	2	1
Logistic Regression	0	1
Single-Layer Perceptron	5	2
Stochastic Gradient Descent	3	2
SVM	4	6
Simple Linear Regression	0	0
Simple Logistic Regression	2	1
Voted Perceptron	1	2
KNN	4	1
K-Star	2	1
Decision Table	4	0
RIPPER	3	1
M5 Rules	3	1
1-R	0	1
PART	2	2
0-R	0	0
Decision Stump	0	0
C4.5 Decision Tree	6	2
Logistic Model Tree	5	2
M5 Tree	3	1
Random Forest	2	3
Random Tree	4	4
REP Tree	2	3
Locally Weighted Learning*	3	0
AdaBoost M1*	2	2
Additive Regression*	1	2
Attribute Selected*	2	0
Bagging*	1	2
Classification via Regression*	0	0
LogitBoost*	4	4
MultiClass Classifier*	3	0
Random Committee*	0	1
Random Subspace*	0	2
Voting+	1	0
Stacking+	0	0

by following paths that only use active hyperparameters. At each node in this traversal, the probability of the corresponding hyperparameter is computed according to its 1-D estimator, and the individual probabilities are combined on a pass back up to the root of the tree. Note that this means that TPE assumes independence for hyperparameters that do not appear together along any path from the tree's root to one of its leaves.

4. AUTO-WEKA

To demonstrate the feasibility of an automatic approach to solving the CASH problem, we built a tool, *Auto-WEKA*, that solves this problem for all classification algorithms and feature selectors/evaluators implemented in the standard WEKA package [14]. Note that while we have focused on classification algorithms in WEKA, there is no obstacle to extending our approach to other settings.

Table 1 provides a list of all 39 WEKA classification algorithms. Of these models, 27 are considered 'base' classifiers (which can be used independently), 10 of the remaining classifiers are meta methods (which take a single base classifier and its parameters as an input), and the final 2 ensemble

Table 2: Feature Search/Evaluator methods in Auto-WEKA. * indicates search methods, which require one feature evaluator that is used to determine the importance of a feature.

Feature Method	Categorical	Numeric
Best First*	1	1
Greedy Stepwise*	3	2
Ranker*	0	1
CFS Subset Eval	2	0
Pearson Correlation Eval	0	0
Gain Ratio Eval	0	0
Info Gain Eval	2	0
1-R Eval	1	2
Principal Components Eval	2	2
RELIEF Eval	1	2
Symmetrical Uncertainty Eval	1	0

classifiers can take any number of base classifiers as input. We allowed the meta-methods to use any base classifier with any hyperparameter settings, and allowed the 2 ensemble methods to use up to five of the 27 base classifiers, again with any hyperparameter settings. Not all classifiers are applicable on all datasets (e.g., due to a classifier's inability to handle missing data). For a given dataset, our Auto-WEKA implementation automatically only considers the subset of applicable classifiers.

Table 2 provides a list of WEKA's 3 feature search methods, as well its 8 feature evaluators, and their respective number of subparameters (up to 5 for search; up to 4 for evaluators). To perform feature selection, a search method is combined with a feature evaluator, and the subparameters of both need to be instantiated. Feature selection is run as a preprocessing phase before building any classifier.

The algorithms in Table 1 and 2 have a wide variety of hyperparameters, which take values from continuous intervals, from ranges of integers, and from other discrete sets. We associated either a uniform or log uniform prior with each numerical parameter, depending on its semantics. For example, we set a log uniform prior for the ridge regression penalty, and a uniform prior for the maximum depth for a tree in a random forest. Auto-WEKA works with continuous hyperparameter values directly up to the precision of the machine; nevertheless, to give a sense of the size of the space we studied, we note that discretizing hyperparameter domains to a maximum of 10 values each gives rise to over 10^{47} hyperparameter settings. We emphasize that this space is *much* larger than a simple union of the base learners' hyperparameter spaces (whose size is roughly 10^8), since the ensemble methods allow up to 5 *independent* base learners, giving rise to a space with roughly $(10^8)^5 = 10^{40}$ elements. Feature selection gives rise to another independent decision between roughly 10^6 choices, and several parameters on the meta and ensemble level contribute another order of magnitude to the total size of AutoWEKA's hyperparameter space.

Auto-WEKA can be understood as a single learning algorithm with a highly conditional hyperparameter space. As depicted in Figure 1, Auto-WEKA has two top-level Boolean parameters. The first is `is_base`, which selects among single base classifiers and ensemble or meta-classifiers. If `is_base` is `true`, then the parameter `base` determines which of the 27 base classifiers are to be used. If `is_base` is `false`, then `class` indicates either an ensemble or a meta-classifier. If `class` is a meta-classifier, then the parameter `meta_base` is chosen to be one of the 27 base classifiers.

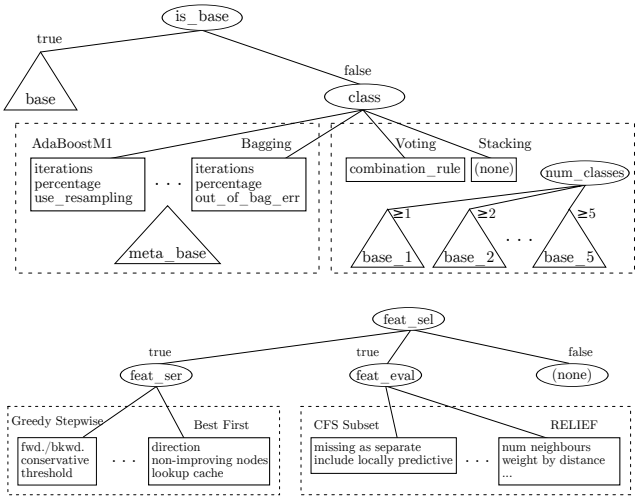

Figure 1: Auto-WEKA's top-level parameters. Top: *is_base* controls Auto-WEKA's classification methods. The triangular items represent a parameter that selects one of the 27 base classifiers, and adds conditional classifier hyperparameters accordingly. Bottom: *feat_sel* controls Auto-WEKA's feature selection methods.

In the event that `class` is an ensemble classifier, an additional parameter `num_classes` is an integer chosen from $\{1, \ldots, 5\}$. `base_i` variables are then selected according to the value of `num_classes`, which again select which of the 27 base classifiers to use. For each of the different `base` parameters, conditional hyperparameters for every model are attached. Auto-WEKA's second top-level Boolean parameter `feat_sel` determines whether to apply one of the feature selection methods. If `feat_sel` is `false`, then Auto-WEKA passes the unmodified dataset to the classifier. If it is true, then `feat_search` selects the choice of feature search method, and `feat_eval` selects the choice of feature evaluator. This results in a very wide tree that captures all the hierarchical nature of the model hyperparameters, and allows the creation of a single hyperparameter optimization problem with four hierarchical layers of a total of 786 parameters.

Auto-WEKA is agnostic to the choice of optimizer, so we implemented variants leveraging SMAC and TPE, respectively.[2] We defined two Auto-WEKA variants, based on SMAC and TPE, respectively. Both of these Auto-WEKA versions are available to the public at `www.cs.ubc.ca/labs/beta/Projects/autoweka`; we are committed to supporting their widespread practical adoption. Both TPE and SMAC have their own parameters that influence their performance (such as TPE's choice of the γ-quantile indicating 'good' or 'bad' performance, or the parameters of SMAC's random forest model). In Auto-Weka, we used the defaults for these meta-hyperparameters, as set by the authors. (Further, small improvements may be obtainable by optimizing these meta-hyperparameters, but a separate process with a meta-training/validation set split would be required to guard against over-fitting, and we did not attempt this). Finally, both TPE and SMAC are randomized algorithms, and thus produce different results based on the random seed provided. As demonstrated in [17], this allows for trivial yet effective parallelization of the optimization process via simply performing k independent runs of the optimization method in parallel and selecting the result of the run with the lowest cross-validation error.[3] We ran Auto-WEKA with 4 such parallel jobs, thereby simulating runs on a standard multicore desktop machine.

[2]We thank the authors of TPE for giving us access to their implementation.

Table 3: Datasets Used; *Num. Discr.*. and *Num. Cont.* refer to the number of discrete and continuous attributes of elements in the dataset, respectively.

Name	Num Discr.	Num Cont.	Num Classes	Num Training	Num Test
Dexter	20 000	0	2	420	180
GermanCredit	13	7	2	700	300
Dorothea	100 000	0	2	805	345
Yeast	0	8	10	1 038	446
Amazon	10 000	0	49	1 050	450
Secom	0	591	2	1 096	471
Semeion	256	0	10	1 115	478
Car	6	0	4	1 209	519
Madelon	500	0	2	1 820	780
KR-vs-KP	37	0	2	2 237	959
Abalone	1	7	28	2 923	1 254
Wine Quality	0	11	11	3 425	1 469
Waveform	0	40	3	3 500	1 500
Gisette	5 000	0	2	4 900	2 100
Convex	0	784	2	8 000	50 000
CIFAR-10-Small	3 072	0	10	10 000	10 000
MNIST Basic	0	784	10	12 000	50 000
Rot. MNIST + BI	0	784	10	12 000	50 000
Shuttle	9	0	7	43 500	14 500
KDD09-Appentency	190	40	2	35 000	15 000
CIFAR-10	3 072	0	10	50 000	10 000

5. EVALUATING AUTO-WEKA

We now describe an experimental study of the performance that can be achieved by Auto-WEKA on various datasets. After specifying our experiment environment, we demonstrate the importance of addressing the algorithm selection and CASH problems, and establish baselines for them (Section 5.2). We evaluate Auto-WEKA's ability to search its enormous hyperparameter space effectively to find algorithms and hyperparameters with low cross-validation error (Section 5.3). Then, we analyze its test performance and address concerns regarding overfitting (Section 5.4). Finally, we provide a synopsis of the classifiers and feature search/evaluators Auto-WEKA chose in our experiments (Section 5.5).

5.1 Experimental setup

We evaluated Auto-WEKA on 21 prominent benchmark datasets (see Table 3): 15 sets from the UCI repository [12]; the 'convex', 'MNIST basic' and 'rotated MNIST with background images' tasks used in [5]; the appentency task from the KDD Cup '09; and two versions of the CIFAR-10 image classification task [20] (CIFAR-10-Small is a subset of CIFAR-10, where only the first 10 000 training data points are used rather than the full 50 000.) For datasets with a predefined training/test split, we used that split. Otherwise, we randomly split the dataset into 70% training and 30% test data. We withheld the test data from all optimization method; it was only used once in an offline analysis stage to evaluate the models found by the various optimization

[3]Other, more sophisticated methods for the parallelization of Bayesian optimization exist [17, 4, 11, 29], but to date, there is no empirical evidence that these methods outperform the simple approach we use here when the cost of evaluating hyperparameter configurations varies across the space.

methods. We denote datasets with at least 10 000 training data points as 'large' and all others as 'small'.

All of our experiments were run on Linux machines with Intel Xeon X5650 six-core processors, running at 2.66GHz. We enforced a RAM limit of 3GB for classification; if training a classifier ever exceeded this memory limit, the classifier job was terminated, returning a misclassification rate of 100%. An additional 1GB of RAM was allocated for the SMBO method. We chose these limits to be reasonably close to the resource limitations faced by a typical user of machine learning algorithms. We also limited the training time for each evaluation of a learning algorithm on each fold, to ensure that the optimization method had a chance to explore the search space. Once this training budget for a fold is consumed, Auto-WEKA sends an interrupt to the learning algorithm to terminate as soon as possible, and the (partially) trained model is then evaluated on the validation set to determine the error estimate of the fold. This timeout was set to 150 minutes for classification and 15 minutes for feature search and evaluation in our experiments.[4] For each dataset, we ran Auto-WEKA with each hyperparameter optimization algorithm with a total time budget of 30 hours. For each method, we performed 25 runs of this process with different random seeds and then—in order to simulate parallelization on a typical workstation—used bootstrap sampling to repeatedly select 4 random runs and report the performance of the one with best cross-validation performance.

In early experiments, we observed a few cases in which Auto-WEKA's SMBO method picked hyperparameters that had excellent training performance, but turned out to generalize poorly. To enable Auto-WEKA to detect such overfitting, we partitioned its training set into two subsets: 70% for use inside the SMBO method, and 30% of validation data that we only used after the SMBO method finished.

5.2 Baseline Methods

Auto-WEKA aims to aid non-expert users of machine learning techniques. A natural approach that such a user might take is to perform 10-fold cross validation on the training set for each technique with unmodified hyperparameters, and select the classifier with the smallest average misclassification error across folds. We will refer to this method applied to the set of 39 WEKA classifiers as *Ex-Def*; it is the best choice that can be made among the 39 WEKA classifiers (with their default hyperparameters) based on exhaustive cross-validation. However, another (unfortunately) common approach for classifier selection is simply to choose based on popularity or intuitive appeal, without any empirical consideration of alternatives. For each dataset, the second and third columns in Table 4 present the best and worst "oracle performance" of the 39 default classifiers when prepared given all the training data and evaluated on the test set. We observe that the gap between the best and worst classifier was huge, *e.g.* misclassification rates of 4.93% *vs* 99.24% on the Dorothea dataset. Even when the set of classifiers was restricted to a few popular ones (we considered neural networks, random forests, SVMs, AdaBoost, C4.5 decision trees, logistic regression, and KNN), this gap still exceeded 20% on 14 out of the 21 datasets. Furthermore, there was no single method that achieved good performance across all datasets: every method was at least 22% worse than the best for at least one data set. This suggests that some form of algorithm selection is essential for achieving good performance. We note that the oracle best performance for Ex-Def provides a lower bound on the classification error that can be achieved via any method that performs only algorithm selection from the 39 WEKA classifiers with default hyperparameter settings).

More experienced users of machine learning algorithms would not only select between a fixed set of default algorithms, but would also consider different hyperparameter settings—for example by performing a grid search over the hyperparameter space of a single classifier (as, e.g., implemented in WEKA).[5] Since different learning algorithms perform well for different problems, users would optimally also want to consider different hyperparameter settings for more than one learning algorithm. Therefore, a stronger baseline we will use is an approach that—in addition to the 39 WEKA default classifiers—considers various hyperparameter settings for all of WEKA's 27 base classifiers. More precisely, this baseline performs an exhaustive search over a grid of hyperparameter settings for each of these 27 base classifiers (plus the 39 WEKA default classifiers), discretizing numeric parameters into three points. We refer to this baseline as *grid search* and note that—as an optimization approach in the joint space of algorithms and hyperparameter settings—it is a simple CASH algorithm. However, it is quite expensive, requiring more than 10 000 CPU hours on each of Gisette, Convex, MNIST, Rot MNIST + BI, and both CIFAR variants, rendering it infeasible to use in most practical applications. (In contrast, we gave Auto-WEKA only 120 CPU hours.)

Table 4 (columns 4 and 5) shows the best and worst "oracle performance" on the test set across the classifiers evaluated by grid search. Comparing these performances to the default performance obtained using Ex-Def, we note that in most cases, even WEKA's best default algorithm could be improved by selecting better hyperparameter settings, sometimes rather substantially: *e.g.*, in the CIFAR-10 small task, grid search offered a 13% reduction in error over Ex-Def.

It has been demonstrated in previous work that, holding the overall time budget constant, grid search is outperformed by random search over the hyperparameter space [5]. Our final baseline, *random search*, implements such a method, picking algorithms and hyperparameters sampled at random, and computes their performance on the 10 cross-validation folds until it exhausts its time budget. For each dataset, we first used 750 CPU hours to compute the cross-validation performance of randomly sampled combinations of algorithms and hyperparameters. We then simulated runs of random search by sampling combinations without replacement from these results that consumed 120 CPU hours and returning the sampled combination with the best performance.

5.3 Results for Cross-Validation Performance

With 786 hierarchical hyperparameters, Auto-WEKA's combined algorithm / hyperparameter space is very complex. We now study how effectively SMAC and TPE searched this space to optimize 10-fold cross-validation performance, and compare their performance to that of Ex-Def, grid search and random search. The middle portion of Table 4 reports our main results. First, we note that grid search over the

[4] In preliminary experiments, only few models exceeded this timeout for the datasets studied here. [28] presents a promising approach for using runtime predictions in the expected improvement calculation to automatically drive the search away from excessively expensive models. We plan to incorporate this approach into future versions of Auto-WEKA.

[5] See WEKA's CVParameterSelection class; `weka.wikispaces.com/Optimizing+parameters`.

Table 4: Performance on both 10-fold cross-validation and test data. Ex-Def and Grid Search are deterministic. Random search had a time budget of 120 CPU hours. For SMAC and TPE, we performed 25 runs of 30 hours each. We report results as mean error rate across 100 000 bootstrap samples simulating 4 parallel runs. We determined test error rates by training the selected model/hyperparameters on the entire 70% training data and computing accuracy on the previously unused 30% test data. Boldface indicates the lowest error within a block of comparable methods that was statistically significant. SC denotes correlation coefficients (see Section 5.4).

Dataset	Oracle Perf. (%)				10-Fold C.V. Performance (%)					Test Performance (%)					SC	
	Ex-Def		Grid Search		Ex-Def	Grid Search	Rand. Search	Auto-WEKA		Ex-Def	Grid Search	Rand. Search	Auto-WEKA		TPE	SMAC
	Best	Worst	Best	Worst				TPE	SMAC				TPE	SMAC		
Dexter	7.78	52.78	**3.89**	63.33	10.20	**5.07**	10.60	9.83	5.66	8.89	**5.00**	9.18	8.89	7.49	0.82	0.25
GermanCredit	26.00	38.00	**25.00**	68.00	22.45	20.20	20.15	21.26	**17.87**	27.33	**26.67**	29.03	27.54	28.24	0.31	0.20
Dorothea	4.93	99.24	**4.64**	99.24	6.03	6.73	8.11	6.81	**5.62**	6.96	5.80	**5.22**	6.15	6.21	0.95	0.40
Yeast	40.00	68.99	**36.85**	69.89	39.43	39.71	38.74	**35.01**	35.51	40.45	42.47	43.15	**40.10**	40.67	0.36	0.49
Amazon	28.44	99.33	**17.56**	99.33	43.94	**36.88**	59.85	50.26	47.34	28.44	**20.00**	41.11	36.59	33.99	0.92	0.97
Secom	7.87	14.26	**7.66**	92.13	6.25	6.12	5.24	6.21	**5.24**	8.09	8.09	8.03	8.10	**8.01**	-0.10	-0.56
Semeion	8.18	92.45	**5.24**	92.45	6.52	4.86	6.06	6.76	**4.78**	8.18	6.29	6.10	8.26	**5.08**	0.84	0.73
Car	0.77	29.15	**0.00**	46.14	2.71	0.83	**0.53**	0.91	0.61	0.77	0.97	**0.01**	0.18	0.40	0.12	0.75
Madelon	**17.05**	50.26	**17.05**	62.69	25.98	26.46	27.95	24.25	**20.70**	21.38	21.15	24.29	21.56	**21.12**	0.44	0.43
KR-vs-KP	0.31	48.96	**0.21**	51.04	0.89	0.64	0.63	0.43	**0.30**	0.31	1.15	0.58	0.54	**0.31**	0.22	0.32
Abalone	73.18	84.04	**72.15**	92.90	73.33	72.15	72.03	72.14	**71.71**	73.18	73.42	74.88	**72.94**	73.51	0.15	0.10
Wine Quality	36.35	60.99	**32.88**	99.39	38.94	35.23	35.36	35.98	**34.65**	37.51	34.06	34.41	**33.56**	33.95	0.73	0.85
Waveform	14.27	68.80	**13.47**	68.80	12.73	12.45	12.43	12.55	**11.92**	14.40	14.66	14.27	**14.23**	14.42	0.36	0.26
Gisette	2.52	50.91	**1.81**	51.23	3.62	2.59	4.84	3.55	**2.43**	2.81	2.40	4.62	3.94	**2.24**	0.69	0.79
Convex	25.96	50.00	**19.94**	71.49	28.68	**22.36**	33.31	28.56	25.93	25.96	23.45	31.20	25.59	**23.17**	0.98	0.84
CIFAR-10-Small	65.91	90.00	**52.16**	90.36	66.59	**53.64**	67.33	58.41	58.84	65.91	56.94	66.12	57.01	**56.87**	0.93	0.80
MNIST Basic	5.19	88.75	**2.58**	88.75	5.12	**2.51**	5.05	10.02	3.75	5.19	**2.64**	5.05	12.28	3.64	1.00	0.87
Rot. MNIST + BI	63.14	88.88	**55.34**	93.01	66.15	**56.01**	68.62	73.09	57.86	63.14	57.59	66.40	70.20	**57.04**	0.50	0.95
Shuttle	0.0138	20.8414	**0.0069**	89.8207	0.0328	0.0361	0.0345	0.0251	**0.0224**	0.0138	0.0414	0.0157	0.0145	**0.0130**	0.60	0.73
KDD09-Appentency	1.7400	6.9733	**1.6332**	54.2400	1.8776	1.8735	1.7510	1.8776	**1.7038**	1.7405	1.7400	1.7400	1.7381	**1.7358**	0.89	1.00
CIFAR-10	64.27	90.00	**55.27**	90.00	65.54	**54.04**	69.46	67.73	62.36	64.27	63.13	69.72	66.01	**61.15**	0.33	0.69

hyperparameters of all base-classifiers yielded better results than Ex-Def in 17/21 cases, which underlines the importance of not only choosing the right algorithm but of also setting its hyperparameters well. However, we note that we gave grid search a very large time budget (often in excess 10 000 CPU hours for each dataset, in total more than 10 CPU years), meaning that it would often be infeasible to use in practice. In contrast, we gave each of the other methods only 4×30 CPU hours per dataset; nevertheless, they still yielded substantially better performance than grid search, outperforming it in 14/21 cases. Random search outperforms grid search in 9/21 cases, highlighting that even exhaustive grid search with a large time budget is not always the right thing to do. Comparing the two Auto-WEKA variants, SMAC outperforms TPE in 19/21 cases. We note that sometimes Auto-WEKA's performance improvements over the baselines were substantial, with relative reductions of the cross-validation error rate exceeding 10% in 6/21 cases.

5.4 Results for Test Performance

The results just shown demonstrate that Auto-WEKA is effective at optimizing its given objective function; however, this is not sufficient to allow us to conclude that it fits models that generalize well. As the number of hyperparameters of a machine learning algorithm grows, so does its potential for overfitting. The use of cross-validation substantially increases Auto-WEKA's robustness against overfitting, but since its hyperparameter space is much larger than that of standard classification algorithms, it is important to carefully study whether (and to what extent) overfitting poses a problem.

To evaluate generalization, we determined a combination of algorithm and hyperparameter settings A_λ by running Auto-WEKA as before (cross-validating on the training set), trained A_λ on the entire training set, and then evaluated the resulting model on the test set. The right portion of Table 4 reports the test performance obtained with all methods. Broadly speaking, similar trends held as for cross-validation performance: Auto-WEKA outperforms the baselines, with grid search and random search performing better than Ex-Def. However, the performance differences were less pronounced: grid search only yields better results than Ex-Def in 15/21 cases, and random search in turn outperforms grid search in 7/21 cases. Auto-WEKA outperforms the baselines in 15/21 cases. Notably, on 12 of the 13 largest datasets, Auto-WEKA outperforms our baselines; we attribute this to the fact that the risk of overfitting decreases with dataset size. Sometimes, Auto-WEKA's performance improvements over the other methods were substantial, with relative reductions of the test error rate exceeding 16% in 3/21 cases. Comparing the different Auto-WEKA variants, SMAC outperformed TPE in 14 cases, and TPE performed better than SMAC in 7. We note that the differences in error rate between SMAC and TPE were typically small, but in the 3 cases with a substantial gap, SMAC produced models with lower classification error. Finally, we note that the CASH problem can also be solved by sophisticated methods based on principles other than Bayesian optimization. In particular, we also evaluated the irace package [22], given the same CPU time as Auto-WEKA. SMAC performed better than irace in 18/21 cases with respect to cross-validation performance. As above, the performance differences were less pronounced for test performance, but SMAC still performed better in 13/21 cases; for the largest sets, where cross-validation performance is more correlated with test set performance, SMAC outperformed irace in 11/13 cases.

As mentioned earlier, Auto-WEKA only used 70% of its training set during the optimization of cross-validation performance, reserving the remaining 30% for assessing the risk of overfitting. At any point in time, Auto-WEKA's SMBO method keeps track of its *incumbent* (the hyperparameter configuration with the lowest cross-validation error rate seen so far). After its SMBO procedure has finished, Auto-WEKA extracts a trajectory of these incumbents from it and computes their generalization performance on the withheld 30%

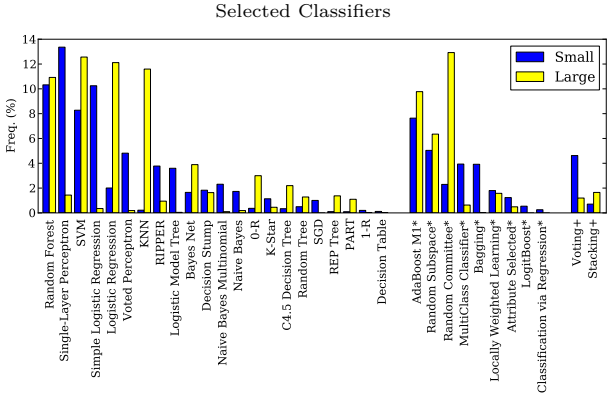

Figure 2: Distribution of chosen classifiers across the small and large datasets, aggregated across TPE, and SMAC, ranked on their frequency of being selected. Meta-methods are marked by a * suffix, ensemble methods by a + suffix.

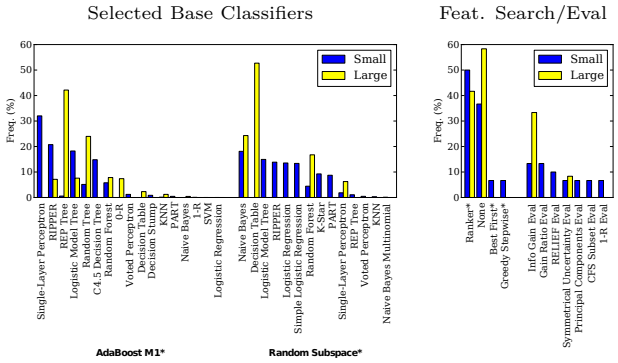

Figure 3: Left: distribution of chosen base classifiers for the two most frequently selected meta-methods: AdaBoostM1 and random subspace. Right: distribution of chosen feature search and evaluator methods. Both plots are aggregated across TPE and SMAC, ranked on their frequency of being selected; *None* indicates that no feature selection was performed.

validation data. It then computes the Spearman rank coefficient between the sequence of training performances (evaluated by the SMBO method through cross-validation) and this generalization performance. The rightmost columns in Table 4 (labelled SC) show the average correlation coefficient for each run of Auto-WEKA. We note a general trend: as the absolute gap between cross-validation and test performance grows, this correlation coefficient decreases. The German-Credit dataset is a good example where Auto-WEKA can signal that it only has low confidence in how well its chosen hyperparameters will generalize. We do note, however, that this weak signal has to be used with caution: there is no guarantee that large correlation coefficients yield a small gap and vice versa.

5.5 Classifiers Selected by Auto-WEKA

Figure 2 shows the distribution of classifiers chosen by our two Auto-WEKA variants (aggregated across runs and datasets - both TPE and SMAC produce similar results when considered individually). We note that no single classifier clearly dominated the others: the most frequently used classifiers (random forests, the single layer perceptron, and SVMs) were only selected in roughly 12% of all cases each, and most classifiers were selected in at least a few percent of the cases. Furthermore, the selected methods differed considerably between the large and small datasets, demonstrating the need for dataset-specific methods; for example, the large datasets benefitted more from meta-methods than the small ones. A more detailed investigation of the top two meta-methods in Figure 3 (left) shows which base methods were chosen. Note that AdaBoostM1 frequently used the single layer perceptron on small datasets, but never on large ones, while the REP tree was often chosen for large datasets. In the random subspace, the two most prominent methods were naive Bayes and the decision table. It is interesting to note that these two methods, as well as the REP tree frequently selected by AdaBoost, were not often selected as base classifiers on their own. This underlines the importance of searching Auto-WEKA's entire parameter space instead of, *e.g.*, restricting one's attention to a small number of favourite base classifiers.

Figure 3 (right) provides a breakdown of the feature search and evaluation methods Auto-WEKA selected. Overall, it used these feature selection methods more often on the smaller datasets than on the larger ones, and if it did use a feature selection method it favored the ranker method. All feature evaluators were used with roughly the same frequency for small datasets; in contrast, if Auto-WEKA performed feature selection for a large dataset it favored the information gain evaluator. We note that Auto-WEKA's data-dependent choices (based on its internal cross-validation evaluation) allow it to use feature selection as a regularization method for small data sets, while at the same time using all features to construct more complex trained models for large datasets.

6. CONCLUSION AND FUTURE WORK

In this work, we have shown that the daunting problem of combined algorithm selection and hyperparameter optimization (CASH) can be solved by a practical, fully automated tool. This is made possible by recent Bayesian optimization techniques that iteratively build models of the algorithm/hyperparameter landscape and leverage these models to identify new points in the space that deserve investigation. We built a tool, Auto-WEKA, that draws on the full range of classification algorithms in WEKA and makes it easy for non-experts to build high-quality classifiers for given application scenarios. An extensive empirical comparison on 21 prominent datasets showed that Auto-WEKA often outperformed standard algorithm selection and hyperparameter optimization methods, especially on large datasets. We empirically compared two different optimizers for searching Auto-WEKA's 786-dimensional parameter space and in the end recommend an Auto-WEKA variant based on the Bayesian optimization method SMAC [15]. We have written a freely downloadable software package to make Auto-WEKA easy for end-users to access; it is available at `www.cs.ubc.ca/labs/beta/Projects/autoweka/`.

We see several promising avenues for future work. First, Auto-WEKA still shows larger improvements in cross-validation performance than on test data, suggesting the investigation of more sophisticated methods for detecting and avoiding overfitting than our simple correlation-based approach. Second, we see potential value in extending our current approach to allow parameter sharing between classifiers used within ensemble methods, likely increasing their chance of being selected by Auto-WEKA. Finally, we could use our approach as an inner loop for training ensembles of machine learning

algorithms by iteratively adding algorithms with maximal marginal contribution. (This idea is conceptually related to the Hydra approach for constructing algorithm selectors [33].)

References

[1] M. Adankon and M. Cheriet. Model selection for the LS-SVM. application to handwriting recognition. *Pattern Recognition*, 42(12):3264–3270, 2009.

[2] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag. Collaborative hyperparameter tuning. In *Proc. of ICML-13*, 2013.

[3] Y. Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8):1889–1900, 2000.

[4] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. In *Proc. of NIPS-11*, 2011.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13:281–305, 2012.

[6] A. Biem. A model selection criterion for classification: Application to HMM topology optimization. In *Proc. of ICDAR-03*, pages 104–108. IEEE, 2003.

[7] H. Bozdogan. Model selection and Akaike's information criterion (AIC): The general theory and its analytical extensions. *Psychometrika*, 52(3):345–370, 1987.

[8] P. Brazdil, C. Soares, and J. Da Costa. Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

[9] E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Technical Report UBC TR-2009-23 and arXiv:1012.2599v1, Department of Computer Science, University of British Columbia, 2009.

[10] O. Chapelle, V. Vapnik, and Y. Bengio. Model selection for small sample regression. *Machine Learning*, 2001.

[11] T. Desautels, A. Krause, and J. Burdick. Parallelizing exploration-exploitation tradeoffs with gaussian process bandit optimization. In *Proc. of ICML-12*, 2012.

[12] A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL: http://archive.ics.uci.edu/ml. University of California, Irvine, School of Information and Computer Sciences.

[13] X. Guo, J. Yang, C. Wu, C. Wang, and Y. Liang. A novel LS-SVMs hyper-parameter selection based on particle swarm optimization. *Neurocomputing*, 71(16):3211–3215, 2008.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[15] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, pages 507–523, 2011.

[16] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *JAIR*, 36(1):267–306, 2009.

[17] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Parallel algorithm configuration. In *Proc. of LION-6*, pages 55–70, 2012.

[18] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[19] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of IJCAI-95*, pages 1137–1145, 1995.

[20] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[21] R. Leite, P. Brazdil, and J. Vanschoren. Selecting classification algorithms with active testing. In *Proc. of MLDM-12*, pages 117–131, 2012.

[22] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[23] O. Maron and A. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Proc. of NIPS-94*, pages 59–66, 1994.

[24] A. McQuarrie and C. Tsai. *Regression and time series model selection*. World Scientific, 1998.

[25] B. Pfahringer, H. Bensusan, and C. Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proc. of ICML-00*, pages 743–750, 2000.

[26] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber. PyBrain. *JMLR*, 2010.

[27] M. Schonlau, W. J. Welch, and D. R. Jones. Global versus local search in constrained optimization of computer models. In N. Flournoy, W. Rosenberger, and W. Wong, editors, *New Developments and Applications in Experimental Design*, volume 34, pages 11–25. Institute of Mathematical Statistics, Hayward, California, 1998.

[28] J. Snoek, H. Larochelle, and R. Adams. Opportunity cost in Bayesian optimization. In *NIPS Workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits*, 2011. Published online.

[29] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proc. of NIPS-12*, 2012.

[30] N. Srinivas, A. Krause, S. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proc. of ICML-10*, pages 1015–1022, 2010.

[31] V. Strijov and G. Weber. Nonlinear regression model generation using hyperparameter optimization. *Computers & Mathematics with Applications*, 60(4):981–988, 2010.

[32] R. Vilalta and Y. Drissi. A perspective view and survey of meta-learning. *Artif. Intell. Rev.*, 18(2):77–95, Oct. 2002.

[33] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proc. of AAAI-10*, pages 210–216, 2010.

[34] P. Zhao and B. Yu. On model selection consistency of lasso. *JMLR*, 7:2541–2563, Dec. 2006.