Surrogate Benchmarks for Hyperparameter Optimization

Katharina Eggensperger[1] and **Frank Hutter**[1] and **Holger H. Hoos**[2] and **Kevin Leyton-Brown**[2]

Abstract. Since hyperparameter optimization is crucial for achieving peak performance with many machine learning algorithms, an active research community has formed around this problem in the last few years. The evaluation of new hyperparameter optimization techniques against the state of the art requires a set of benchmarks. Because such evaluations can be very expensive, early experiments are often performed using synthetic test functions rather than using real-world hyperparameter optimization problems. However, there can be a wide gap between the two kinds of problems. In this work, we introduce another option: cheap-to-evaluate surrogates of real hyperparameter optimization benchmarks that share the same hyperparameter spaces and feature similar response surfaces. Specifically, we train regression models on data describing a machine learning algorithm's performance under a wide range of hyperparameter configurations, and then cheaply evaluate hyperparameter optimization methods using the model's performance predictions in lieu of the real algorithm. We evaluate the effectiveness for using a wide range of regression techniques to build these surrogate benchmarks, both in terms of how well they predict the performance of new configurations and of how much they affect the overall performance of hyperparameter optimizers. Overall, we found that surrogate benchmarks based on random forests performed best: for benchmarks with few hyperparameters they yielded almost perfect surrogates, and for benchmarks with more complex hyperparameter spaces they still yielded surrogates that were qualitatively similar to the real benchmarks they model.

1 Introduction

The performance of many machine learning methods depends critically on hyperparameter settings and thus on the method used to set such hyperparameters. Recently, sequential model-based Bayesian optimization methods, such as SMAC[16], TPE[2], and Spearmint[29] have been shown to outperform more traditional methods for this problem (such as grid search and random search [3]) and to rival—and in some cases surpass—human domain experts in finding good hyperparameter settings [29, 30, 5]. One obstacle to further progress in this nascent field is a paucity of reproducible experiments and empirical studies. Until recently, a study introducing a new hyperparameter optimizer would typically also introduce a new set of hyperparameter optimization benchmarks, on which the optimizer would be demonstrated to achieve state-of-the-art performance (as compared to, e.g., human domain experts). The introduction of the hyperparameter optimization library (HPOlib [8]), which offers a unified interface to different optimizers and benchmarks, has made it easier to reuse previous benchmarks and to systematically compare different approaches [4].

[1] University of Freiburg, email:{eggenspk,fh}@cs.uni-freiburg.de
[2] University of British Columbia, email: {hoos,kevinlb}@cs.ubc.ca

However, a substantial problem remains: performing a hyperparameter optimization experiment requires running the underlying machine learning algorithm, often at least hundreds of times. This is infeasible in many cases. The first (mundane, but often significant) obstacle is to get someone else's research code working on one's own system—including resolving dependencies and acquiring required software licenses—and to acquire the appropriate input data. Furthermore, some code requires specialized hardware; most notably, general-purpose graphics processing units (GPGPUs) have become a standard requirement for the effective training of modern deep learning architectures [20, 21]. Finally, the computational expense of hyperparameter optimization can be prohibitive for research groups lacking access to large compute clusters. These problems represent a considerable barrier to the evaluation of new hyperparameter optimization algorithms on the most challenging and interesting hyperparameter optimization benchmarks, such as deep belief networks [2], convolutional neural networks [29, 5], and combined model selection and hyperparameter optimization in machine learning frameworks [30].

Given this high overhead for studying complex hyperparameter optimization benchmarks, most researchers have drawn on simple, synthetic test functions from the global continuous optimization community [12]. While these are simple to use, they are often poorly representative of the hyperparameter optimization problem: in contrast to the response surfaces of actual such problems, these synthetic test functions are smooth and often have unrealistic shapes. Furthermore, they only involve real-valued parameters and hence do not incorporate the categorical and conditional parameters typical of actual hyperparameter optimization benchmarks.

In the special case of small, finite hyperparameter spaces, a much better alternative is simply to record the performance of every hyperparameter configuration, thereby speeding future evaluations via a table lookup. The result is a perfect *surrogate* of an algorithm's true performance that takes time $O(1)$ to compute (using a hash) and that can be used in place of actually running the algorithm and evaluating its performance. This table-based surrogate can trivially be transported to any new system, without the complicating factors involved in running the original algorithm (setup, special hardware requirements, licensing, computational cost, etc.). In fact, several researchers have already applied this approach to simplifying their experiments: for example, Bardenet et al. [1] saved the performances of a parameter grid with 108 points of Adaboost on 29 datasets, and Snoek et al. [29] saved the performance of parameter grids with 1400 and 288 points for a structured SVM [31] and an online LDA [13], respectively. The latter two benchmarks are part of HPOlib and are, in fact, HPOlib's most frequently used benchmarks, due to their simplicity of setup and low computational cost.

Of course, the drawback of this table lookup idea is that it is limited

to small, finite hyperparameter spaces. Here, we generalize the idea of machine learning algorithm surrogates to arbitrary, potentially high-dimensional hyperparameter spaces (including, e.g., real-valued, categorical, and conditional hyperparameters). As in the table-lookup strategy, we first evaluate many hyperparameter configurations during an expensive offline phase. We then use the resulting performance data to train a regression model to approximate future evaluations via model predictions. As before, we obtain a surrogate of algorithm performance that is cheap to evaluate and trivially portable. However, model-based surrogates offer only *approximate* representations of performance. Thus, a key component of our work presented in the following is an investigation of the quality of these approximations.

We are not the first to propose the use of learned surrogate models that stand in for computationally complex functions. In the field of metalearning [6], regression models have been extensively used to predict the performance of algorithms across various datasets based on dataset features [11, 26]. The statistics literature on the design and analysis of computer experiments (DACE) [27, 28] uses similar surrogate models to guide a sequential experimental design strategy aiming to achieve either an overall strong model fit or to identify the minimum of a function. Similarly, the SUrrogate MOdeling (SUMO) Matlab toolkit[10] provides an environment for building regression models to describe the outputs of expensive computer simulations based on active learning. Such an approach for finding the minimum of a blackbox function also underlies the sequential model-based Bayesian optimization framework [7, 16] (SMBO, the framework underlying all hyperparameter optimizers we study here). While all of these lines of work incrementally construct surrogate models of a function in order to inform an active learning criterion that determines new inputs to evaluate, our work differs in its goals: We train surrogates on a set of data gathered offline (by some arbitrary process—in our case the combination of many complete runs of several different SMBO methods plus random search) and use the resulting surrogates as stand-in models for the entire hyperparameter optimization benchmark.

The *surrogate benchmarks* resulting from our work can be used in several different ways. Firstly, like synthetic test functions and table lookups, they can be used for extensive debugging and unit testing. Since the large computational expense of running hyperparameter optimizers is typically dominated by the cost of evaluating algorithm performance under different selected hyperparameters, our benchmarks can also substantially reduce the time required for running a hyperparameter optimizer, facilitating whitebox tests of an optimizer using exactly the hyperparameter space of the machine learning algorithm whose performance is modelled by the surrogate. This functionality is gained even if the surrogate model only fits algorithm performance quite poorly (e.g., due to a lack of sufficient training data). Finally, a surrogate benchmark whose model fits algorithm performance very well can also facilitate the evaluation of new features inside the hyperparameter optimizer, or even the (meta-)optimization of a hyperparameter optimizer's own hyperparameters (which can be useful even without the use of surrogates, but is typically extremely expensive [17]).

The rest of this paper is laid out as follows. We first provide some background on hyperparameter optimization (Section 2). Then, we discuss our methodology for building surrogate benchmarks (Section 3) using several types of machine learning models. Next, we evaluate the performance of these surrogates in practice (Section 4). We demonstrate that random forest models tend to fit the data better than a broad range of competing models, both in terms of raw predictive model performance and in terms of the usefulness of the resulting surrogate benchmark for comparing hyperparameter optimization procedures.

2 Background: Hyperparameter Optimization

The construction of machine learning models typically gives rise to two optimization problems. The first is internal optimization, such as selecting a neural network's likelihood-maximizing weights; the second is tuning the method's hyperparameters, such as setting a neural network's regularization parameters or number of neurons. The former problem is closely coupled with the machine learning algorithm at hand and is very well studied; here, we consider the latter. Let $\lambda_1, \ldots, \lambda_n$ denote the hyperparameters of a given machine learning algorithm, and let $\Lambda_1, \ldots, \Lambda_n$ denote their respective domains. The algorithm's hyperparameter space is then defined as $\boldsymbol{\Lambda} = \Lambda_1 \times \cdots \times \Lambda_n$. When trained with hyperparameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ on data $\mathcal{D}_{\text{train}}$, the algorithm's loss (e.g., misclassification rate) on data $\mathcal{D}_{\text{valid}}$ is $\mathcal{L}(\boldsymbol{\lambda}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}})$. Using k-fold cross-validation, the optimization problem is then to minimize:

$$f(\boldsymbol{\lambda}) = \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(\boldsymbol{\lambda}, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}). \quad (1)$$

A hyperparameter λ_n can have one of several types, such as continuous, integer-valued or categorical. For example, the learning rate for a neural network is continuous; the random seed given to initialize an algorithm is integer-valued; and the choice between various preprocessing methods is categorical. Furthermore, there can be *conditional* hyperparameters, which are only active if another hyperparameter takes a certain value; for example, the hyperparameter "number of principal components" only needs to be instantiated when the hyperparameter "preprocessing method" is PCA.

Evaluating $f(\boldsymbol{\lambda})$ for a given $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ is computationally costly, and so many techniques have been developed to find good configurations $\boldsymbol{\lambda}$ with few function evaluations. The methods most commonly used in practice are manual search and grid search, but recently, it has been shown that even simple random search can yield much better results [3]. The state of the art in practical optimization of hyperparameters is defined by Bayesian optimization methods [16, 29, 2], which have been successfully applied to problems ranging from deep neural networks to combined model selection and hyperparameter optimization [2, 29, 30, 19, 5].

Bayesian optimization methods use a probabilistic model $\mathcal{M}$ to model the relationship between a hyperparameter configuration $\boldsymbol{\Lambda}$ and its performance $f(\boldsymbol{\lambda})$. They fit this model using previously gathered data and then use it to select a next point $\boldsymbol{\lambda}_{\text{new}}$ to evaluate, trading off exploitation and exploration in order to find the minimum of f. They then evaluate $f(\boldsymbol{\lambda}_{\text{new}})$, update $\mathcal{M}$ with the new data $(\boldsymbol{\lambda}_{\text{new}}, f(\boldsymbol{\lambda}_{\text{new}}))$ and iterate. Throughout this paper, we will use the following three instantiations of Bayesian optimization:

SPEARMINT [29] is a prototypical Bayesian optimization method that models $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ with Gaussian process (GP) models. It supports continuous and discrete parameters (by rounding), but no conditional parameters.

Sequential Model-based Algorithm Configuration (SMAC) [16] models $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ with random forests. When performing cross validation, SMAC only evaluates as many folds as necessary to show that a configuration is worse than the best one seen so far (or to replace it). SMAC can handle continuous, categorical, and conditional parameters.

Tree Parzen Estimator (TPE) [2] models $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ indirectly. It models $p(f < f^*)$, $p(\boldsymbol{\lambda} \mid f < f^*)$, and $p(\boldsymbol{\lambda} \mid f \geq f^*)$, where f^* is defined as a fixed quantile of the function values observed so

far, and the latter two probabilities are defined by tree-structured Parzen density estimators. TPE can handle continuous, categorical, and conditional parameters.

An empirical evaluation on the three methods on the HPOlib hyperparameter optimization benchmarks showed that SPEARMINT performed best on benchmarks with few continuous parameters and SMAC performed best on benchmarks with many, categorical, and/or conditional parameters, closely followed by TPE. SMAC also performed best on benchmarks that relied on cross-validation [8].

3 Methodology

We now discuss our approach, including the algorithm performance data we used, how we preprocessed the data, the types of regression models we evaluated, and how we used them to construct surrogate benchmarks.

3.1 Data collection

In principle, we could construct surrogate benchmarks using algorithm performance data gathered by any means. For example, we could use existing data from a manual exploration of the hyperparameter space, or from an automated approach, such as grid search, random search or one of the more sophisticated hyperparameter optimization methods discussed in Section 2.

It is more important for surrogate benchmarks to exhibit strong predictive quality in some parts of the hyperparameter space than in others. Specifically, our ultimate aim is to ensure that *hyperparameter optimizers perform similarly on the surrogate benchmark as on the real benchmark*. Since most optimizers spend most of their time in high-performance regions of the hyperparameter space, and since relative differences between the performance of hyperparameter configurations in such high-performance regions tend to impact which hyperparameter configuration will ultimately be returned, accuracy in this part of the space is more important than in regions of poor performance. The training data should therefore densely sample high-performance regions. We thus advocate collecting performance data primarily via runs of existing hyperparameter optimization procedures. As an additional advantage of this strategy, we can obtain this costly performance data as a by-product of executing hyperparameter optimization procedures on the original benchmark.

Of course, it is also important to accurately identify poorly performing parts of the space: if we only trained on performance data for the very best hyperparameter settings, no machine learning model could be expected to infer that performance in the remaining parts of the space is poor. This would typically lead to underpredictions of performance in poor parts of the space. We thus also included performance data gathered by a random search. (An alternative is grid search, which can also cover the entire space. We did not adopt this approach because it cannot deal effectively with large hyperparameter spaces.) To gather the data for each surrogate benchmark in this paper, we therefore executed $r = 10$ runs of each of the three Bayesian optimization methods described in Section 2 (each time with a different seed), as well as random search, with each run gathering the performance of a fixed number of configurations.

3.2 Data preprocessing

For each benchmark we studied for this paper, after running the hyperparameter optimizers and random search, we preprocessed the data as follows:

Table 1. Overview of evaluated regression algorithms. When we used random search to optimize hyperparameters, we considered 100 samples over the stated hyperparameters (their names refer to the SCIKIT-LEARN implementation [25]); the model was trained on 50% of the data, and the best configuration was chosen based on the performance on the other 50% and then trained on all data.

Model	**Hyperparameter optimization**	Impl.
Random Forest	None	[25]
Gradient Boosting	None	[25]
Extra Trees	None	[25]
Gaussian Process	MCMC sampling over hyperparameters	[29]
SVR	Random search for `C` and `gamma`	[25]
NuSVR	Random search for `C`, `gamma` and `nu`	[25]
Bayesian Neural Network	None	[24]
k-nearest-neighbours	Random search for `n_neighbors`	[25]
Linear Regression	None	[25]
Least Angle Regression	None	[25]
Ridge Regression	None	[25]

1. We extracted all available configuration/performance pairs from the runs. For benchmarks that used cross-validation, we encoded the cross-validation fold of each run as an additional categorical parameter (for benchmarks without cross validation, that parameter was set to a constant).
2. We removed entries with invalid results caused by algorithm crashes. Since some regression models used in preliminary experiments could not handle duplicated configurations, we also deleted these, keeping the first occurrence.
3. For data from benchmarks featuring conditional parameters, we replaced the values of inactive conditional parameters with a default value.
4. To code categorical parameters, we used a one-hot (aka 1-in-k) encoding, which replaces any single categorical parameter λ with domain $\Lambda = \{k_1, \dots k_n\}$ by n binary parameters, only the i-th of which is true for data points where λ is set to k_i.

3.3 Choice of Regression Models

We considered a broad range of commonly used regression algorithms as candidates for our surrogate benchmarks. To keep the results comparable, all models were trained on data encoded as detailed in the previous section. If necessary for the algorithm, we also normalized the data to have zero mean and unit variance (by subtracting the mean and dividing by the standard deviation). If not stated otherwise for a model, we used the default configuration of its implementation.

Table 1 details the regression models and implementations we used. We evaluated three different tree-based models, because SMAC uses a random forest (RF), and because RFs have been shown to yield high-quality predictions of algorithm performance data [18]. As a specialist for low-dimensional hyperparameter spaces, we used SPEARMINT's Gaussian process (GP) implementation, which performs MCMC to marginalize over hyperparameters. Since SMAC performs particularly well on high-dimensional hyperparameter spaces and SPEARMINT on low-dimensional continuous problems [8], we expected their respective models to mirror that pattern. The remaining prominent model types we experimented with comprised k-nearest-neighbours (kNN), linear regression, least angle regression, ridge regression, SVM methods (all as implemented by scikit-learn [25]), and Bayesian neural networks (BNN) [24].

Table 2. Properties of our data sets. "Input dim." is the number of features of the training data; it is greater than the number of hyperparameters because categorical hyperparameters and the crossvalidation fold are one-hot-encoded. For each benchmark, before preprocessing the number of data points was $10 \times 4\times$ (#evals. per run).

	hyperparameter			Input	#evals.	#data
	#λ	cond.	cat. / cont.	dim.	per run	
Branin	2	-	- / 2	3	200	7402
Log. Reg. 5CV	4	-	- / 4	9	500	18521
HP-NNET convex	14	4	7 / 7	25	200	7750
HP-DBNET mrbi	36	27	19 / 17	82	200	7466

3.4 Construction and Use of Surrogate Benchmarks

To construct surrogates for a hyperparameter optimization benchmark X, we trained the previously mentioned models on the performance data gathered on benchmark X. The surrogate benchmark X'_M based on model M is identical to the original benchmark X, except that evaluations of the machine learning algorithm to be optimized in benchmark X are replaced by a performance prediction obtained from model M. In particular, the surrogate's configuration space (including all parameter types and domains) and function evaluation budget are identical to the original benchmark.

Importantly, the wall clock time to run an algorithm on X'_M is much lower than that required on X, since expensive evaluations of the machine learning algorithm underlying X are replaced by cheap model predictions. The model M is simply saved to disk and is queried when needed. We could implement each evaluation in X'_M as loading M from disk and then using it for prediction, but to avoid the repeated cost of loading M, we also allow for storing M in an independent process and communicate with it via a local socket.

To evaluate the performance of a surrogate benchmark scenario X'_M we ran the same optimization experiments as on X, using the same settings and seeds. In addition to evaluating the raw predictive performance of model M, we assessed the quality of surrogate benchmark X'_M by measuring the similarity of hyperparameter optimization performance on X and X'_M.

4 Experiments and Results

In this section, we experimentally evaluate the performance of our surrogates. We describe the data upon which our surrogates are based, evaluate the raw performance of our regression models on this data, and then evaluate the quality of the resulting surrogate benchmarks.

4.1 Experimental Setup

We collected data for four benchmarks from the hyperparameter optimization benchmark library, HPOLIB [8]. For each benchmark, we executed 10 runs of SMAC, SPEARMINT, TPE and random search (using the same Hyperopt implementation of random search as for TPE), yielding the data detailed in Table 2. The four benchmarks comprised two low-dimensional and two high-dimensional hyperparameter spaces.

The two low-dimensional benchmarks were the synthetic Branin test function and a logistic regression [29] on the MNIST dataset [23]. Both of these have been extensively used before to benchmark hyperparameter optimization methods. While the 2-dimensional Branin test function is trivial to evaluate and therefore does not require a surrogate, we nevertheless included it to study how closely a surrogate can approximate the function. The logistic regression is an actual hyperparameter optimization benchmark with 4 hyperparameters that includes a 5-fold cross-validation. That means for each configuration that the optimizers TPE, SPEARMINT and random search evaluated there were 5 data points that only differ in which fold they corresponded to. Since SMAC saves time by not evaluating all folds for configurations that appear worse than the optimum, it only evaluated a subset of folds for most of the configurations. The evaluation of a single cross-validation fold required roughly 1 minute on a single core of an Intel Xeon E5-2650 v2 CPU.

The high-dimensional benchmarks comprised a simple and a deep neural network, HP-NNET and HP-DBNET (both taken from [2]) to classify the MRBI and convex datasets, respectively [22]. Their dimensionalities are 14 and 36, respectively, and many categorical hyperparameters further increase the input dimension to the regression model. Evaluating a single HP-NNET configuration required roughly 12 minutes using 2 cores of an Intel Xeon E5-2650 v2 with OpenBlas. The HP-DBNET required a GPGPU to run efficiently; on a modern Geforce GTX780 GPU, it took roughly 15 minutes to evaluate a single configuration. In contrast, using the surrogate benchmark model we built, one configuration can be evaluated in less than a second on a standard CPU.

For some model types, training with all the data from Table 2 was computationally infeasible, and we had to subsample 2 000 data points (uniformly at random[3]) for training. This was the case for nuSVR, SVR, and the Bayesian neural network. For the GP model, we had to limit the dataset even further to 1 500 data points. On this reduced training set, the GP model required 255 minutes to train on the most expensive data set (HP-DBNET MRBI), and the Bayesian neural networks required 36 minutes; all other models required less than one minute for training.

We used HPOLIB to run the experiments for all optimizers with a single format, both for the original hyperparameter optimization benchmarks and for our surrogates. To make our results reproducible, we fixed the pseudo-random number seed in each function evaluation to 1. The version of the SPEARMINT package we used crashed for about 1% of all runs due to a numerical problem. In evaluations where we require entire trajectories, for these crashed SPEARMINT runs, we imputed the best function value found before the crash for all evaluations after the crash.

4.2 Evaluation of Raw Model Performance

We first studied the raw predictive performance of the models we considered on our preprocessed data.

4.2.1 Using all data

To evaluate the raw predictive performance of the models listed in Table 1, we used 5-fold cross-validation performance and computed the cross-validated root mean squared error (RMSE) and Spearman's rank correlation coefficient (CC) between model predictions and the true responses in the test fold. Here, the responses correspond to validation error rate in all benchmarks except for the Branin one (where they correspond to the value of the Branin function).

Table 3 presents these results, showing that the GP and the SVR approaches performed best on the smooth low-dimensional synthetic Branin test function, but that RF-based models are better for predicting the performance of actual machine learning algorithms. This

[3] For a given dataset and fold, all models based on the same number of data points used the same subsampled data set. We note that model performance sometimes was quite noisy with respect to the pseudorandom number seed for this subsampling step and we thus used a fixed seed.

Table 3. Average RMSE and CC for a 5-fold cross validation for different regression models. For each entry, bold face indicates the best performance on this dataset, and underlined values are not statistically significantly different from the best according to a paired t-test (with $p = 0.05$). Models marked with an * ($^+$) are trained on only a subset of 2000 (1500) data points per fold.

Model	Branin RMSE	Branin CC	Log.Reg. 5CV RMSE	Log.Reg. 5CV CC	HP-NNET convex RMSE	HP-NNET convex CC	HP-DBNET mrbi RMSE	HP-DBNET mrbi CC
RF	1.86	1.00	**0.03**	**0.98**	0.04	0.94	**0.06**	**0.90**
Gr.Boost	7.10	0.96	0.07	0.94	0.05	0.89	0.06	0.86
Ex.Trees	1.10	1.00	0.04	0.98	**0.03**	**0.95**	0.06	0.90
GP $^+$	0.03	**1.0**	0.13	0.88	0.04	0.92	0.10	0.78
SVR *	0.06	1.0	0.13	0.87	0.06	0.82	0.08	0.80
nuSVR *	**0.02**	1.0	0.18	0.84	0.06	0.85	0.08	0.82
BNN *	6.84	0.91	0.10	0.91	0.05	0.84	0.10	0.72
kNN	1.78	1.00	0.14	0.88	0.06	0.85	0.08	0.78
Lin.Reg.	45.33	0.28	0.23	0.78	0.08	0.60	0.1	0.70
LeastAngleReg.	45.33	0.28	0.23	0.78	0.08	0.60	0.1	0.7
Ridge Reg.	46.01	0.30	0.26	0.77	0.09	0.61	0.10	0.67

Table 4. Average RMSE and CC of 5 regression algorithms in the leave-one-optimizer-out setting. Bold face indicates the best value across all regression models on this dataset.

Model	Branin RMSE	Branin CC	Log.Reg. 5CV RMSE	Log.Reg. 5CV CC	HP-NNET convex RMSE	HP-NNET convex CC	HP-DBNET mrbi RMSE	HP-DBNET mrbi CC
RF	2.04	0.95	**0.10**	**0.93**	**0.04**	0.82	**0.07**	**0.85**
Gr.Boost	6.96	0.85	0.12	0.84	0.05	0.81	**0.07**	0.83
GP	0.12	**0.99**	0.16	0.76	0.05	**0.84**	0.10	0.64
nuSVR	**0.03**	0.98	0.16	0.74	0.10	0.61	0.10	0.73
kNN	2.08	0.96	0.19	0.72	0.07	0.73	0.09	0.67

strong performance was to be expected for the higher-dimensional hyperparameter space of the neural networks, since RFs perform automatic feature selection.[4] The logistic regression example is rather low-dimensional, but the categorical cross-validation fold is likely harder to model for GPs than for RFs.[5] Extra Trees predicted the performance of the actual machine learning algorithms nearly as good as the RF, Gradient boost was slightly worse. Bayesian neural networks, k-nearest-neighbours and our linear regression models could not achieve comparable performance. Based on these results, we decided to focus the remainder of our study on a diverse subset of models: two tree-based approaches (RFs and gradient boost), Gaussian processes, nuSVR, and, as an example of a popular, yet poorly-performing model, k-nearest-neighbours. We paid special attention to RFs and Gaussian processes, since these have been used most prominently in Bayesian hyperparameter optimization methods.

4.2.2 *Leave one optimizer out*

In practice, we will want to use our surrogate models to predict the performance of a machine learning algorithm with hyperparameter configurations selected by some new optimization method. The configurations it evaluates might be quite different from those considered by the optimizers whose data we trained on. Next to the standard cross-validation setting from above, we therefore evaluated our models in the leave-one-optimizer-out setting, which means that the regression model learns from data drawn from all but one optimizer, and its performance is measured on the held out data.

Table 4 reports RMSE and CC analogous to those of Table 3, but for the leave-one-optimizer-out setting. This setting is more difficult and, consequently, the results were slightly worse, but the best-performing models stayed the same: nuSVR and GP for low dimensional and RFs for higher dimensional datasets.

[4] We note that RFs could also handle the categorical hyperparameters in these benchmarks natively. We used the one-hot encoding for comparability with other methods. It is remarkable that even with this encoding, they outperformed all other methods.

[5] As we will see later (Figure 2), the RF-based optimizer SMAC also performed better on this benchmark than the GP-based optimizer SPEARMINT.

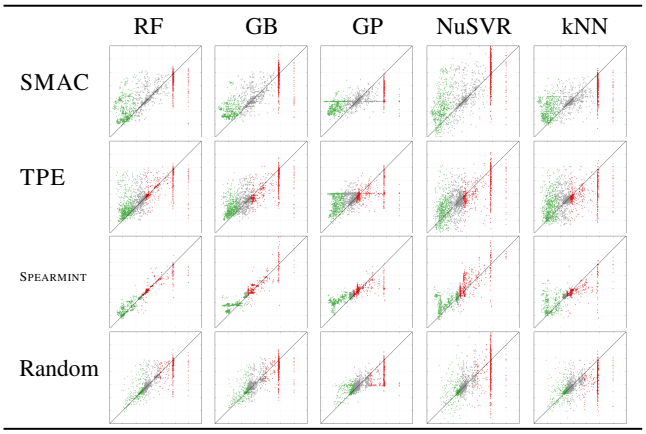

Figure 1. True performance (x-axis) vs. regression model predictions (y-axis) for the **HP-DBNET mrbi** dataset. All plots have the same axes, showing error rates ranging from 0.4 to 1.1. Each marker represents the performance of one configuration; green and red crosses indicate 1/3 best and worst true performance, respectively. Configurations on the diagonal are predicted perfectly, error predictions above the diagonal are too high, and predictions for configurations below the diagonal are better than the configuration's actual performance. The first column shows which data was left out for training and used for testing.

Figure 1 studies the predictive performance in more detail for the HP-DBNET mrbi benchmark, demonstrating that tree-based models also performed best in a qualitative sense. The figure also shows that the models tended to make the largest mistakes for the worst configurations; especially the non-tree-based models predicted some of these to be better than some of the best configurations. The same patterns also held for the logistic regression 5CV and the HP-NNET convex data (not shown). The models also completely failed to identify neural network configurations which did not converge within the time limit and therefore received an error rate of 1.0. Interestingly, the GP failed almost entirely on the high-dimensional HP-DBNET MRBI benchmark in two cases, predicting all data points around the data mean.

4.3 Evaluation of Surrogate Benchmarks

We now study the performance of the surrogate benchmarks X'_M obtained for random forest (RF) and Gaussian process (GP) models M. We assess the quality of X'_M by comparing the performance of various hyperparameter optimizers on X'_M and the real benchmark X.

4.3.1 *Using all data*

We first analyzed the performance of surrogate benchmarks based on models trained on the entire data we have available. We note that in this first experiment, a surrogate that perfectly remembers the training data would achieve perfect performance, because we used the same hyperparameter optimizers for evaluation as we did to gather the training data. However, after the first imperfect prediction, the trajectories of the optimizers will diverge. Thus, since none of our models is perfect on training data, this initial experiment serves as an

evaluation of surrogate benchmarks based on training data gathered through the same mechanism as at test time.

We performed experiments for our three actual hyperparameter optimization benchmarks, logistic regression, a simple and a deep neural network. For each of them, we repeated the 10 runs for TPE, SMAC and SPEARMINT we previously conducted on the real benchmarks, but now used surrogate benchmarks based on RFs and GPs, respectively.

Figure 2 shows that the surrogate benchmarks based on Gaussian process models differed substantially from the true benchmarks. The figures show the best function values found by the various optimizers over time. Visually comparing the first column (real benchmark) to the third (surrogate benchmark based on GP model), the most obvious difference is that the surrogate benchmark fails completely on HP-DBNET MRBI: since the GP model is unable to properly fit the high-dimensional data (predicting all configurations to perform roughly equally, around the data mean) all optimizers basically stay at the same performance level (the data mean). Note in the plot for the true benchmark that the GP-based optimizer SPEARMINT also performed very poorly on this benchmark.

In the other two cases (logistic regression 5CV and HP-NNET convex), the performance of the optimizers appears visually similar to the true benchmark at first glance. However, for the logistic regression 5CV the GP model predicts some parts of the hyperparameter space to be better than the actual best part of the space, leading to the final optimization results on the surrogate benchmark to appear better than optimization results on the true benchmark. Another difference is a zig-zag pattern in the trajectories for logistic regression surrogates: these are also (mildly) present in the real benchmark (mild enough to only be detectable when zooming into the figure) and are due to the slightly different performance in the 5 folds of cross validation; the impact of the folds is very small, but the GP model predicts it to be large, causing the zig-zag. Interestingly, for the GP model trained on the HP-NNET convex dataset, regions with "better" performance appear hard to find: only SMAC and TPE identified them, causing a larger gap between SPEARMINT and SMAC/TPE than on the real benchmark.

Conversely, the RF surrogates yielded results much closer to those obtained on the real benchmark. Visually, the first column (true benchmark) and second column appear very similar, indicating that the RF captured the overall pattern well. There are some differences in the details. For example, on HP-NNET convex, the surrogate does not capture that TPE finds very good configurations before SMAC and yields the overall best performance. Nevertheless, overall, our results for the RF surrogates qualitatively resemble those for the true benchmarks, and for the logistic regression example, the correspondence is almost perfect.

4.3.2 Leave one optimizer out

Next, we studied the use of a surrogate benchmark to evaluate a new optimizer. For each optimizer o and each of the three hyperparameter optimization benchmarks X, we trained RF and GP models M on the respective leave-one-optimizer-out training data discussed in Section 4.2.2 and compared the performance of optimizer o on X and X'_M. Figure 3 reports the results of this experiment, showing that surrogate benchmarks based on RF models qualitatively resembled the real benchmarks.

The results for the logistic regression 5CV benchmark (top row of Figure 3) show that surrogate benchmarks based on RF models mirrored the performance of each optimizer o on the real benchmark well, even when the training data did not include data gathered with optimizer o. In contrast, surrogates based on Gaussian process models performed poorly: the Gaussian process again underestimated the error, predicting better performance in some regions than possible on the real benchmark.[6] Again, these regions with "better" performance appear hard to find: only SMAC and SPEARMINT found them, causing their performances on the GP-based surrogate benchmark to differ substantially from their performance on the true benchmark.

Results for HP-NNET convex were also better for the surrogate benchmark based on RFs (especially for SPEARMINT), but not as much better as for the logistic regression 5CV case. As was already the case when the surrogate was based on all training data, the RF-based surrogate benchmarks only approximately captured the strong performance TPE showed on the real benchmark.

Results on HP-DBNET MRBI show a fairly close correspondence between the real benchmark and the RF-based surrogate benchmarks. In contrast, the GP-based surrogate was dismal, once again due to the GP's near-constant predictions (close to the data mean).

After this qualitative evaluation of the surrogate benchmarks, Table 5 offers a quantitative evaluation. We judge the quality of a surrogate benchmark X'_M by how closely it resembles the real benchmark X it was derived from, in terms of the absolute error between the best found values for our four optimizers (SMAC, TPE, SPEARMINT, and random search) after evaluating i configurations. For logistic regression 5CV, in line with our qualitative results we obtained a very small error for the RF-based surrogate. The GP-based surrogate underestimated the achievable error rates, resulting in larger differences between performances on the true and the surrogate runs. After 50 evaluations the GP-based surrogate trained on all data yielded a quite high error because it underestimated the performance for configurations selected by the optimizer SMAC. Training the GP-surrogate on the leave-one-optimizer-out dataset causes worse performance for the optimizer SPEARMINT and too much variation for SMAC resulting in a higher error as well. This misprediction decreases with more evaluated configurations.

The results for the HP-NNET convex look quite similar, with a somewhat smaller difference between RF-based and GP-based surrogates. Indeed, SMAC and TPE behaved similarly on both RF-based and GP-based surrogates as on the real benchmark; only SPEARMINT behaved very differently on the GP-based surrogate, causing an overall higher error than for the RF-based surrogates.

On the high dimensional HP-NNET mrbi the surrogates performed differently. Whereas the RF-based surrogate could still reproduce similar optimizer behavior as on the real benchmark, the GP completely failed to do so. Remarkably, overall quantitative performance was similar for surrogate benchmarks trained on all data and those trained on leave-one-optimizer-out datasets.

Overall, these results confirmed our expectation from previous findings in Section 3.3 and the raw regression model performance results in Table 3: good regression models facilitate good surrogate benchmarks. In our case, RFs performed best for both tasks. We note that using the surrogate benchmarks reduced the time requirements substantially; for example, evaluating a surrogate 100 times instead of the HP-NNET convex or HP-DBNET MRBI took less than 1 minute on a single CPU, compared to roughly 10 hours on two CPUs (HP-NNET convex) and over a day on a modern GPU (HP-DBNET MRBI).[7]

[6] We noticed similar behavior for the nuSVR, which even returned negative values for configurations and caused the optimizer to search completely different areas of the configuration space (data not shown here).

[7] Of course, the overhead due to the used hyperparameter optimizer comes on top of this; e.g., SPEARMINT's overhead for a run with 200 evaluations was roughly one hour, whereas SMAC's overhead was less than one minute.

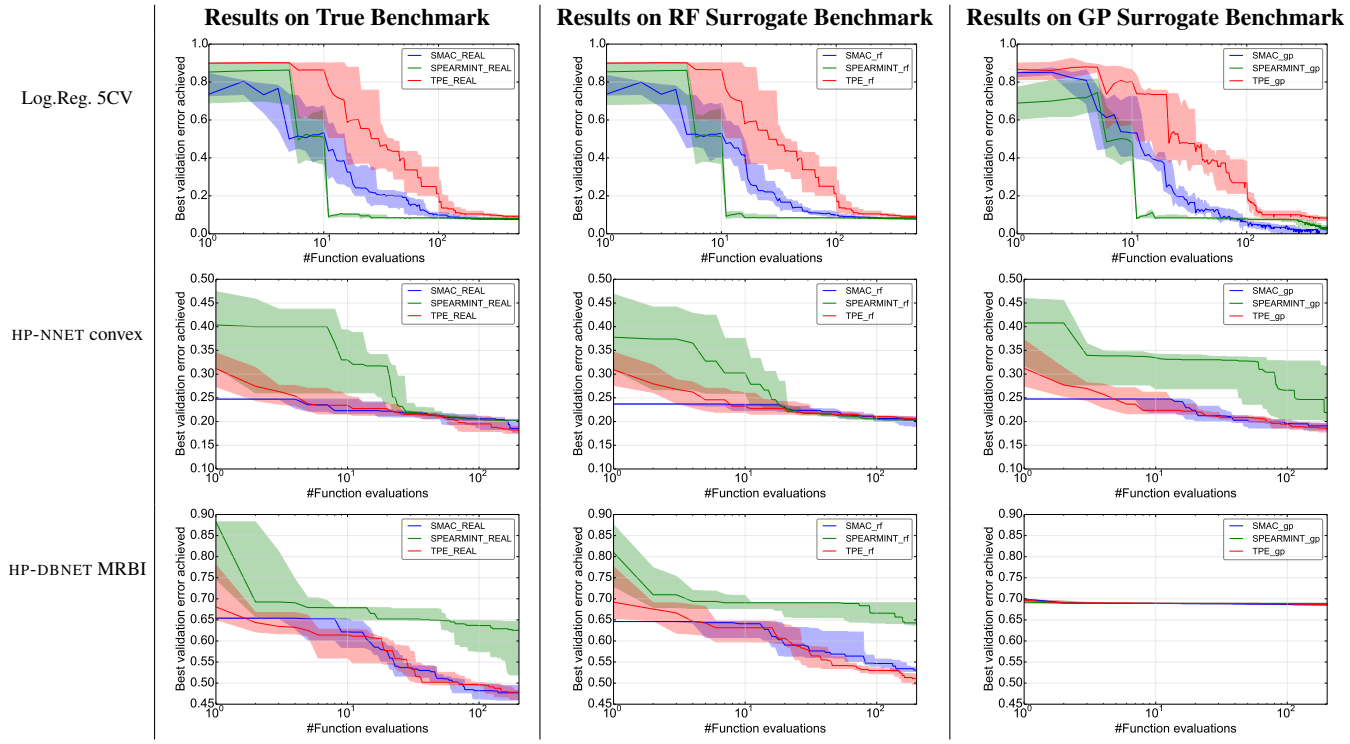

Figure 2. Median and quartile of best performance over time on the real benchmark (left column) and on surrogates (middle: based on RF models; right: based on GP models). Both types of surrogate benchmarks were trained on all available data. For logistic regression 5CV each fold is plotted as a separate function evaluation.

Table 5. Quantitative evaluation of surrogate benchmarks at three different time steps each. We show the mean difference between the best found values for corresponding runs (having the same seed) of the four optimizers (SMAC, TPE, SPEARMINT, and random search) after i function evaluations on the real and surrogate benchmark. For each experiment and optimizer we conducted 10 runs and report the mean error averaged over $4 \times 10 = 40$ comparisons. We evaluated RF-based and GP-based surrogates. For each problem we measured the error for surrogates trained on *all* and the leave-one-optimizer-out (*leave-ooo*) data; e.g., the TPE trajectories are from optimizing on a surrogate that is trained on all training data except that gathered using TPE. Bold face indicates the best performance for this dataset and i function evaluations. Results are underlined when the one-sigma confidence intervals of the best and this result overlaps.

#Function evaluations	50		200		500	
Surrogate	RF	GP	RF	GP	RF	GP
Log.Reg. SCV all	**0.02**	0.06	**0.00**	0.04	**0.00**	0.05
Log.Reg. SCV leave-ooo	**0.02**	0.07	**0.01**	0.03	**0.00**	0.03

#Function evaluations	50		100		200	
Surrogate	RF	GP	RF	GP	RF	GP
HP-NNET convex all	**0.01**	0.03	**0.01**	0.03	**0.01**	0.02
HP-NNET convex leave-ooo	**0.02**	0.03	**0.02**	0.04	**0.02**	0.03
HP-DBNET MRBI all	**0.05**	0.13	**0.05**	0.16	**0.05**	0.17
HP-DBNET MRBI leave-ooo	**0.04**	0.13	**0.04**	0.16	**0.05**	0.17

5 Conclusion and Future Work

To tackle the high computational cost and overhead of performing hyperparameter optimization benchmarking, we proposed surrogate benchmarks that behave similarly to the actual benchmarks they are derived from, but are far cheaper and simpler to use. The key idea is to collect (configuration, performance) pairs from the actual benchmark and to learn a regression model that can predict the performance of a new configuration and therefore stand in for the expensive-to-evaluate algorithm. These surrogates reduce the algorithm overhead to a minimum, which allows extensive runs and analyses of new hyperparameter optimization techniques. We empirically demonstrated that we can obtain surrogate benchmarks that closely resemble the real benchmarks they were derived from.

In future work, we intend to study the use of surrogates for general algorithm configuration. In particular, we plan to support optimization across a set of problem instances, each of which can be described by a fixed-length vector of characteristics, and to assess the resulting surrogates for several problems that algorithm configuration has tackled successfully, such as propositional satisfiability [14], mixed integer programming [15], and AI planning [9]. Finally, good surrogate benchmarks should enable us to explore the configuration options of the optimizers themselves, and we plan to use surrogate benchmarks to enable efficient meta-optimization of the hyperparameter optimization and algorithm configuration methods themselves.

REFERENCES

[1] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag, 'Collaborative hyperparameter tuning', in *Proc. of ICML'13*, (2013).

[2] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, 'Algorithms for hyper-parameter optimization', in *Proc. of NIPS'11*, (2011).

[3] J. Bergstra and Y. Bengio, 'Random search for hyper-parameter optimization', *JMLR*, **13**, 281–305, (2012).

[4] J. Bergstra, B. Komer, C. Eliasmith, and D. Warde-Farley, 'Preliminary evaluation of hyperopt algorithms on HPOLib', in *ICML workshop on AutoML*, (2014).

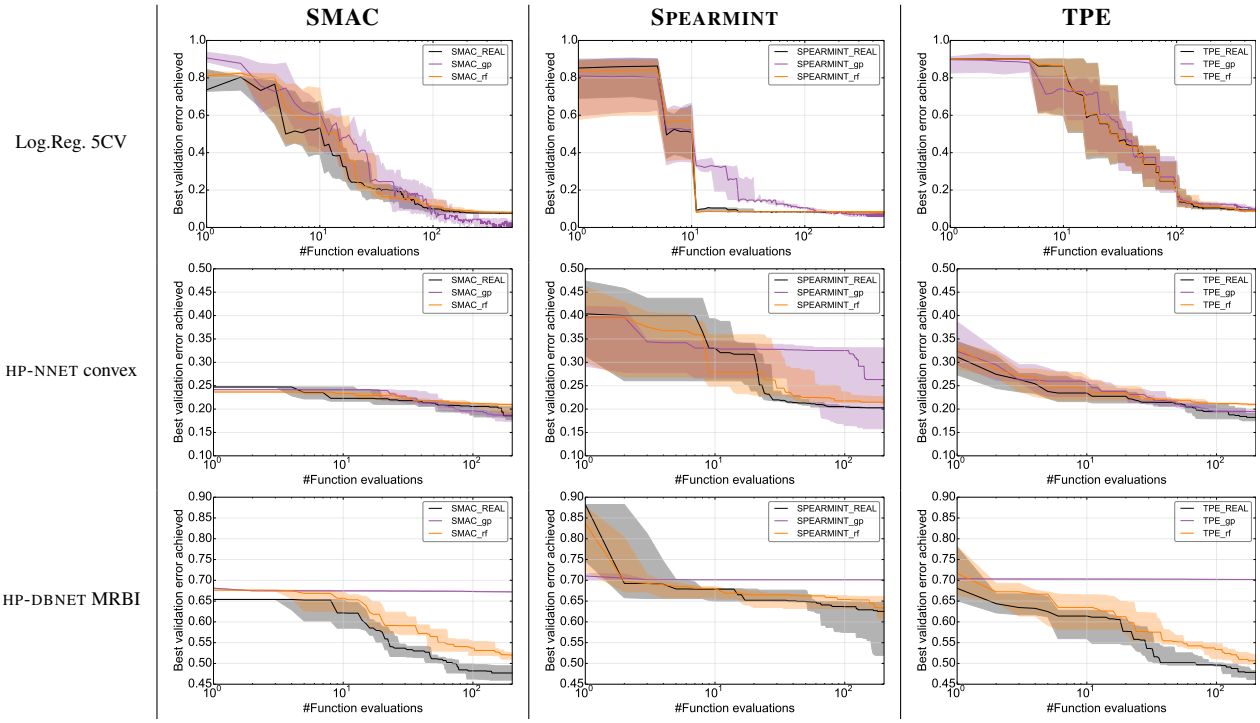

Figure 3. Median and quartile of optimization trajectories for surrogates trained in the leave-one-optimizer-out setting. Black trajectories correspond to true benchmarks, coloured trajectories to optimization runs on a surrogate. The first row names the optimizer used to obtain the trajectories; their data was left out for training the regression models.

[5] J. Bergstra, D. Yamins, and D. D. Cox, 'Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures', in *Proc. of ICML'13*, (2013).

[6] P. Brazdil, C. Giraud-Carrier, C. Soares, and R. Vilalta, *Metalearning: Applications to Data Mining*, Springer, 2008.

[7] 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', *CoRR*, **abs/1012.2599**, (2010).

[8] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. H. Hoos, and K. Leyton-Brown, 'Towards an empirical foundation for assessing bayesian optimization of hyperparameters', in *NIPS workshop on Bayesian Optimization*, (2013).

[9] C. Fawcett, M. Helmert, H. H. Hoos, E. Karpas, G. Röger, and J. Seipp, 'FD-Autotune: Domain-specific configuration using fast-downward', in *Proc. of ICAPS-PAL*, (2011).

[10] D. Gorissen, I. Couckuyt, P. Demeester, T. Dhaene, and K. Crombecq, 'A surrogate modeling and adaptive sampling toolbox for computer based design', *JMLR*, **11**, 2051–2055, (2010).

[11] S. B. Guerra, R. B. C. Prudêncio, and T. B. Ludermir, 'Predicting the performance of learning algorithms using support vector machines as meta-regressors', in *Proc. of ICANN'08*, volume 5163, pp. 523–532, (2008).

[12] N. Hansen, A. Auger, S. Finck, R. Ros, et al. Real-parameter black-box optimization benchmarking 2010: Experimental setup, 2010.

[13] M. D. Hoffman, D. M. Blei, and F. R. Bach, 'Online learning for latent dirichlet allocation.', in *Proc. of NIPS'10*, (2010).

[14] F. Hutter, D. Babić, H.H. Hoos, and A.J. Hu, 'Boosting Verification by Automatic Tuning of Decision Procedures', in *Proc. of FMCAD'07*, pp. 27–34, Washington, DC, USA, (2007). IEEE Computer Society.

[15] F. Hutter, H. H. Hoos, and K. Leyton-Brown, 'Automated configuration of mixed integer programming solvers', in *Proc. of CPAIOR-10*, pp. 186–202, (2010).

[16] F. Hutter, H. H. Hoos, and K. Leyton-Brown, 'Sequential model-based optimization for general algorithm configuration', in *Proc. of LION-5*, (2011).

[17] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, 'ParamILS: an automatic algorithm configuration framework', *JAIR*, **36**(1), 267–306, (2009).

[18] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, 'Algorithm runtime prediction: Methods and evaluation', *JAIR*, **206**(0), 79 – 111, (2014).

[19] B. Komer, J. Bergstra, and C. Eliasmith, 'Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn', in *ICML workshop on AutoML*, (2014).

[20] A. Krizhevsky, 'Learning multiple layers of features from tiny images', Technical report, University of Toronto, (2009).

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, 'Imagenet classification with deep convolutional neural networks', in *Proc. of NIPS'12*, pp. 1097–1105, (2012).

[22] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, 'An empirical evaluation of deep architectures on problems with many factors of variation', in *Proc. of ICML'07*, (2007).

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, 'Gradient-based learning applied to document recognition', *Proc. of the IEEE*, **86**(11), 2278–2324, (1998).

[24] R. M. Neal, *Bayesian learning for neural networks*, Ph.D. dissertation, University of Toronto, 1995.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, 'Scikit-learn: Machine learning in Python', *JMLR*, **12**, 2825–2830, (2011).

[26] M. Reif, F. Shafait, M. Goldstein, T. Breuel, and A. Dengel, 'Automatic classifier selection for non-experts', *PAA*, **17**(1), 83–96, (2014).

[27] J. Sacks, W. J. Welch, T. J. Welch, and H. P. Wynn, 'Design and analysis of computer experiments', *Statistical Science*, **4**(4), 409–423, (November 1989).

[28] T. J. Santner, B. J. Williams, and W. I. Notz, *The design and analysis of computer experiments*, Springer, 2003.

[29] J. Snoek, H. Larochelle, and R.P. Adams, 'Practical Bayesian optimization of machine learning algorithms', in *Proc. of NIPS'12*, (2012).

[30] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, 'Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms', in *Proc. of KDD'13*, (2013).

[31] C. N. J. Yu and T. Joachims, 'Learning structural svms with latent variables', in *Proc. of ICML'09*, pp. 1169–1176, (2009).