Efficient Benchmarking of Hyperparameter Optimizers via Surrogates

Katharina Eggensperger and Frank Hutter
University of Freiburg
{eggenspk, fh}@cs.uni-freiburg.de

Holger H. Hoos and Kevin Leyton-Brown
University of British Columbia
{hoos, kevinlb}@cs.ubc.ca

Abstract

Hyperparameter optimization is crucial for achieving peak performance with many machine learning algorithms; however, the evaluation of new optimization techniques on real-world hyperparameter optimization problems can be very expensive. Therefore, experiments are often performed using cheap synthetic test functions with characteristics rather different from those of real benchmarks of interest. 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 depending on its hyperparameter setting, and then cheaply evaluate hyperparameter optimization methods using the model's performance predictions in lieu of running the real algorithm. We evaluated a wide range of regression techniques, both in terms of how well they predict the performance of new hyperparameter settings and in terms of the quality of surrogate benchmarks obtained. We found that tree-based models capture the performance of several machine learning algorithms well and yield surrogate benchmarks that closely resemble real-world benchmarks, while being much easier to use and orders of magnitude cheaper to evaluate.

Introduction

The performance of many machine learning methods depends crucially on hyperparameter settings and thus on the method used to set these hyperparameters. Recently, sequential model-based Bayesian optimization methods have been shown to outperform more traditional methods for this problem (such as grid search and random search) and to rival—and in some cases surpass—human domain experts in finding good hyperparameter settings (Snoek, Larochelle, and Adams 2012; Thornton et al. 2013; Bergstra, Yamins, and Cox 2013). Similar successes have been achieved by general algorithm configuration methods on a wide range of problems, such as tuning combinatorial problem solvers for Boolean satisfiability (Hutter et al. 2007) and mixed integer programming (Hutter et al. 2009). Here, however, we focus on the special case of hyperparameter optimization.

One obstacle to further progress in hyperparameter optimization is a paucity of reproducible experiments and empirical studies. The hyperparameter optimization library HPOlib (Eggensperger et al. 2013) represents a first step towards alleviating this problem, by offering a unified interface to different optimizers and benchmarks that makes it easier to reuse previous benchmarks and to systematically compare different approaches.

However, experiments with interesting real-world hyperparameter optimization benchmarks often remain 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 use 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 (Krizhevsky, Sutskever, and Hinton 2012). Finally, the computational expense of comprehensive hyperparameter optimization experiments can be prohibitive for research groups lacking access to large compute clusters. These problems represent a considerable barrier to the sound evaluation of new hyperparameter optimization algorithms on the most challenging and interesting hyperparameter optimization benchmarks, such as deep belief networks (Bergstra et al. 2011), convolutional neural networks (Snoek, Larochelle, and Adams 2012; Bergstra, Yamins, and Cox 2013), and combined model selection and hyperparameter optimization in machine learning frameworks (Thornton et al. 2013).

Given this large overhead for studying complex hyperparameter optimization benchmarks, researchers have used simple synthetic test functions, such as the Branin function, to compare hyperparameter optimization algorithms (Snoek, Larochelle, and Adams 2012). While such functions are cheap to evaluate, they are not representative of real hyperparameter optimization problems. In contrast to the response surfaces of the latter, 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 many real 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 up future evaluations via table lookup. This table-based *surrogate*

can be trivially transported to any new system, without whatever complicating factors were involved in running the original algorithm (setup, special hardware requirements, licensing, computational cost, etc.). In fact, several researchers have already applied this approach to simplify experiments (Bardenet et al. 2013; Snoek, Larochelle, and Adams 2012; Birattari et al. 2002).

Unfortunately, table lookup is limited to small, finite hyperparameter spaces. Here, we generalize the idea of such surrogates to arbitrary, potentially high-dimensional hyperparameter spaces (including, e.g., real-valued, categorical, and conditional hyperparameters). As with table lookup, we first evaluate many hyperparameter configurations in an expensive offline phase. Departing from this paradigm, we then use the resulting performance data to train a regression model that approximates future evaluations via model predictions. As before, we obtain a surrogate of algorithm performance that is cheap to evaluate and trivially portable. Since these model-based surrogates offer only *approximate* representations of performance, it is crucial to investigate the quality of their predictions, as we do in this work.[1]

We are not the first to propose the use of learned surrogate models that stand in for computationally complex functions. In the field of meta-learning (Brazdil et al. 2008), regression models have been extensively used to predict the performance of algorithms across various datasets based on dataset features (Guerra, Prudêncio, and Ludermir 2008; Reif et al. 2014). The statistics literature on the design and analysis of computer experiments (DACE) (Sacks et al. 1989; Santner, Williams, and Notz 2003; Gorissen et al. 2010) 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. Surrogate models are also at the core of the sequential model-based Bayesian optimization framework (Brochu, Cora, and de Freitas 2010; Hutter, Hoos, and Leyton-Brown 2011) (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 goal: to obtain *surrogate benchmarks* rather than to identify good points in the space.

Surrogate benchmarks are useful in several different senses. First, like synthetic test functions and table lookups, they can be used for extensive debugging and unit testing. Second, 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. This functionality is gained even if the surrogate model fits algorithm performance quite poorly (e.g., due to a lack of sufficient training data). Third, surrogate benchmarks that closely resemble real benchmarks 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 also be done without using surrogates, but is typically extremely expensive (Hutter et al. 2009)).

[1]This paper is an extended and improved version of a paper presented at an ECAI workshop (Eggensperger et al. 2014).

Background: Hyperparameter Optimization

The construction of machine learning models typically gives rise to two optimization problems: internal optimization (such as selecting a neural network's likelihood-maximizing weights) and hyperparameter optimization (such as setting a neural network's structural and regularization parameters). In this work 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 $\mathbf{\Lambda} = \Lambda_1 \times \cdots \times \Lambda_n$. When trained with hyperparameters $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ 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 the expression

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

A hyperparameter λ_n can be continuous, integer-valued or categorical. For example, the learning rate for a neural network is continuous; the number of neurons is integer-valued; and the choice between various preprocessing methods is categorical. Hyperparameters can also be *conditional*, meaning that their values only matter 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 $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ is often computationally costly; thus, many techniques have been developed to find good configurations $\boldsymbol{\lambda}$ without performing many function evaluations. The methods most commonly used in practice are manual search and grid search, but recently, it was shown that even simple random search can yield much better results (Balaprakash, Birattari, and Stützle 2007; Bergstra and Bengio 2012).

The state of the art in practical optimization of hyperparameters is set by Bayesian optimization methods (Hutter, Hoos, and Leyton-Brown 2011; Snoek, Larochelle, and Adams 2012; Bergstra et al. 2011), which have been successfully applied to problems ranging from deep neural networks to combined model selection and hyperparameter optimization (Bergstra et al. 2011; Snoek, Larochelle, and Adams 2012; Thornton et al. 2013; Komer, Bergstra, and Eliasmith 2014; Bergstra, Yamins, and Cox 2013). Bayesian optimization methods use a probabilistic model $\mathcal{M}$ to describe the relationship between a hyperparameter configuration $\mathbf{\Lambda}$ and its performance $f(\boldsymbol{\lambda})$. They fit this model using previously gathered data and then use it to select a subsequent configuration $\boldsymbol{\lambda}_{\text{new}}$ 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 Bayesian optimization methods:

– **SPEARMINT** (Snoek, Larochelle, and Adams 2012) models $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ as a Gaussian process (GP). It supports continuous and discrete parameters (albeit only via rounding), but not conditional parameters.

– **Sequential Model-based Algorithm Configuration (SMAC)** (Hutter, Hoos, and Leyton-Brown 2011) models $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ as a random forest. 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. SMAC can handle continuous, categorical, and conditional parameters.

– Tree Parzen Estimator (TPE) (Bergstra et al. 2011) models $p_{\mathcal{M}}(f \mid \boldsymbol{\lambda})$ indirectly using tree-structured Parzen density estimators. TPE can handle continuous, categorical, and conditional parameters.

An empirical evaluation of the three methods on the HPOlib 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 (Eggensperger et al. 2013).

Methodology

We now describe the algorithm performance data we used, the types of regression models we evaluated, and how we used them to construct surrogate benchmarks. To avoid confusion, we explicitly state that there are three different types of models: the *base learner* (whose hyperparameters are optimized), the *internal surrogate model* of the hyperparameter optimizer (SMAC, TPE, or Spearmint), and the *benchmark simulator surrogate model*.

Data collection

In principle, we could construct surrogate benchmarks using algorithm performance data gathered by any means, but of course, we prefer to gather data in a way that leads to the best surrogates. 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 effective 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. 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 overly optimistic predictions of performance in poor parts of the space. We therefore also included performance data gathered by random search. (Alternatively, one could use grid search, which can also cover the entire space. We did not adopt this approach, because it cannot deal effectively with large hyperparameter spaces.) Thus, to gather the data for a surrogate of benchmark X, we used the data gathered by empirically evaluating four methods on X: the three previously mentioned Bayesian optimization methods as well as random search.

Table 1: Overview of regression algorithms we evaluated. We used random search to optimize hyperparameters and considered 100 samples over the stated hyperparameters; we trained the model on 50% of the data, chose the best configuration based on its performance on the other 50%, and then trained on all data.

Model	Hyperparameter optimization	Implementation
Gradient Boosting	Random search: `max_features`, `min_samples_leaf`, `max_depth`, `learning_rate`, `n_estimators`	SCIKIT-LEARN
Random Forest	Random search: `min_samples_split`, `n_estimators`, `max_features`	SCIKIT-LEARN
Gaussian Process	MCMC sampling over hyperparameters	SPEARMINT
SVR	Random search: `C` and `gamma`	SCIKIT-LEARN
NuSVR	Random search: `C`, `gamma` and `nu`	SCIKIT-LEARN
k-nearest-neighbours	Random search: `n_neighbors`	SCIKIT-LEARN
Linear Regression	None	SCIKIT-LEARN
Ridge Regression	Random search: `alpha`	SCIKIT-LEARN

Choice of Regression Models

We considered a broad range of commonly used regression algorithms as candidates for our surrogate benchmarks.

Table 1 details the regression models and implementations we used. We considered two different tree-based models, because random forest (RFs) have been shown to yield high-quality predictions of algorithm performance data (Hutter et al. 2014) and because SMAC uses a RF. We also included SPEARMINT's Gaussian process (GP) implementation, which performs MCMC to marginalize over hyperparameters. These models are quite complementary: Spearmint's GPs work best on low-dimensional smooth hyperparameter optimization problems, while SMAC's RFs perform particularly well for non-smooth and high-dimensional problems, such as Auto-WEKA or structure search in deep learning (Eggensperger et al. 2013). As a baseline, we also experimented with k-nearest-neighbours (kNN), linear regression, ridge regression, and two SVM methods (all as implemented by scikit-learn, version 0.15.1 (Pedregosa et al. 2011)).

Construction and Use of Surrogate Benchmarks

To construct surrogates for a hyperparameter optimization benchmark X, we trained each of the previously mentioned models on the performance data gathered by running all four of our hyperparameter optimization methods on benchmark

X. The surrogate benchmark X'_M based on model M is identical to the original benchmark X, except that evaluations of the base learner to be optimized in benchmark X are replaced by a performance prediction[2] 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 all evaluations of the base learner underlying X are replaced by cheap model evaluations. To avoid the repeated cost of training or loading M, we also allow for storing M in an independent process and communicating with it via a local socket.

Experiments and Results

We now present an experimental evaluation of the quality of surrogates constructed by different machine learning methods. Due to limited space, we provide more detailed results in supplementary material: *www.automl.org/benchmarks/aaai2015-surrogates-supplementary.pdf*

Experimental Setup

We experimented with nine benchmarks from the hyperparameter optimization benchmark library HPOLIB (Eggensperger et al. 2013), including three low-dimensional and six high-dimensional hyperparameter spaces. The low-dimensional benchmarks were derived from a logistic regression (Snoek, Larochelle, and Adams 2012) with 4 hyperparameters on the MNIST dataset (LeCun et al. 1998) (both with and without 5-fold cross validation) and an active learning fit of a latent Dirichlet allocation (Hoffman, Blei, and Bach 2010) with 3 continuous hyperparameters. The evaluation of a single configuration of the logistic regression required roughly 1 minute on a single core of an Intel Xeon E5-2650 v2 CPU, whereas the onlineLDA took up to 10 hours. The high-dimensional benchmarks were derived from a simple and a deep neural network, HP-NNET and HP-DBNET (both taken from Bergstra et al. (2011)), each being used to classify the mrbi and the convex datasets (Larochelle et al. 2007). For HP-NNET we also included 5-fold cross validation variants of both datasets. Evaluating a single HP-NNET configuration required roughly 12 minutes using 2 cores with OpenBlas. To run efficiently, the HP-DBNET required a GPGPU; on a modern Geforce GTX780 GPU, it took roughly 15 minutes to evaluate a single configuration.

For each benchmark, we executed 10 runs each of SMAC, SPEARMINT, TPE and random search (using the Hyperopt implementation of both random search and TPE). Table 2 provides an overview of this data. For benchmarks that included 5-fold cross-validation, the data for TPE, SPEARMINT and random search repeats every configuration 5 times, once per fold. In contrast, SMAC natively manages the number of cross-validation folds considered per configuration, and hence evaluated only a subset of folds for most configurations.

[2]Our benchmark algorithms are deterministic, and we thus predicted means, but in principle we could also sample from the predictive distribution produced by a regression model to mimic the behaviour of stochastic algorithms.

Table 2: Properties of our data sets. "One-hot dim." is the number of features in our one-hot encoded training data.

	hyperparameter			One-hot	#evals.	#data
	#λ	cond.	cat. / cont.	dim.	per run	
onlineLDA	3	-	- / 3	4	50	1 999
Log. Reg.	4	-	- / 4	5	100	4 000
Log. Reg. 5CV				9	500	20 000
HP-NNET convex	14	4	7 / 7	25	200	8 000
HP-NNET convex 5CV				29	500	19 998
HP-NNET mrbi	14	4	7 / 7	25	200	8 000
HP-NNET mrbi 5CV				29	500	20 000
HP-DBNET convex	36	27	19 / 17	82	200	7 997
HP-DBNET mrbi						7 916

We used a one-hot (aka 1-in-k) encoding to code categorical parameters and scaled the x values (using the training data) to be within $[0, 1]$. For some model types, training with all the data from Table 2 was computationally infeasible, and so we subsampled 2 000 data points (uniformly at random[3]) for training. This was the case for nuSVR and the GP model. On this reduced training set, the GP model required 550 minutes, and the nuSVR 230 minutes respectively, to train on the most expensive data set (HP-DBNET convex).

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. The version of the SPEARMINT package we used crashed for about 5% 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.

Evaluation of Raw Model Performance

To evaluate the raw predictive performance of the models listed in Table 1, we computed root mean squared error (RMSE) and Spearman's rank correlation coefficient (CC) between model predictions and the true responses.

Using all data Table 3 presents results for 4 representative datasets in a 5-fold cross-validation setting, showing that our tree-based models were best for predicting the performance of our base learners. The RF model achieved the highest CC on all 9 datasets, while the GB model tended to yield the lowest RMSE (details shown in Appendix, Table B.1). Besides the tree-based models, the GP performed best; kNN and the linear regression models did not achieve comparable performance. Based on these results, we decided to focus the remainder of our study on a subset of models: tree-based approaches (RFs and GB), GPs, and, as an example of a popular, yet poorly-performing model, kNN.

[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. To make our results more easily reproducible, we used a fixed seed.

Table 3: Regression performance in the **cross-validation** setting. We report average RMSE and CC for a 5-fold cross validation for different regression models for 4 out of 9 benchmark problem datasets 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 ($p = 0.05$). For models marked with an * we reduced the training data to 2000 data points per fold.

	onlineLDA		Log.Reg.		Log.Reg. 5CV		HP-DBNET mrbi	
Model	RMSE	CC	RMSE	CC	RMSE	CC	RMSE	CC
GB	**29.7**	0.99	**0.061**	0.95	**0.028**	0.96	**0.052**	0.92
RF	32.7	**0.99**	0.068	**0.95**	0.029	**0.98**	0.052	**0.92**
GP*	77.0	0.94	0.114	0.89	0.125	0.88	0.081	0.80
SVR	145.0	0.98	0.124	0.88	0.108	0.89	0.093	0.72
NuSVR*	144.9	0.98	0.131	0.86	0.141	0.86	0.089	0.73
KNN	154.4	0.97	0.146	0.88	0.137	0.89	0.106	0.62
Lin.Reg.	198.7	0.86	0.252	0.66	0.232	0.78	0.094	0.70
Rid.Reg.	198.7	0.86	0.252	0.66	0.232	0.78	0.094	0.70

Leave one optimizer out To evaluate whether our regression models successfully predicted the performance of a machine learning algorithm with hyperparameter configurations selected by some new optimization method we considered a *leave-one-optimizer-out* (*leave-ooo*) setting, learning from data drawn from all but one optimizer and measuring performance on the held-out data. In our experiments we obtained qualitatively similar results for this setting to those of Table 3 (shown in supplementary material). Figure 1 shows representative scatter plots of true vs. predicted performance on test data. For the low-dimensional logistic regression example (first row of Figure 1) the tree-based models and the GP model predicted most configurations very well. Higher-dimensional datasets, such as HP-DBNET mrbi (second row of Figure 1), gave rise to larger errors, with the GP model predicting that many good configurations would perform as poorly as the worst ones.

As RFs and GB performed similarly and RFs are widely used within SMBO, we focus on RFs for the remaining experiments (but provide further details on GB in the supplementary material). We compare RFs to GPs, the best-performing non-tree-based approach.

Evaluation of Surrogate Benchmarks

Having narrowed down the set of machine learning approaches that deserves consideration, we turn to an evaluation of the quality of the surrogate benchmarks produced by our regression models. For benchmark X and model M, we measure the quality of surrogate X'_M by comparing the performance of various hyperparameter optimizers on that surrogate X'_M and the real X. More precisely, we used the leave-one-optimizer-out setting: to benchmark each optimizer, we used a different surrogate, namely one that does not include training data from this optimizer.

Table 4 presents the results of an empirical evaluation using the real benchmarks (left block) versus the surrogate benchmarks based on RFs (middle block) and GPs (right block).[4] Comparing the performance blocks shows that the RF-based surrogate benchmarks yielded results similar to those obtained on the real benchmarks, and that the GP-based surrogate benchmarks gave rise to more different results. For example, for the logistic regression benchmark (with and without cross-validation), the GP surrogate predicted some configurations to have (nonsensical) *negative* loss, and indeed SMAC found these configurations when executed on the GP-based surrogate benchmark. Other results on the GP-based surrogate, such as SPEARMINT's and TPE's performances on the HP-NNET convex benchmark were also too good. In contrast, RFs never predict values higher or lower than contained in the training data, which led them to avoid such errors. In 4 out of 9 benchmarks the best optimizer achieved the same best result as on the real benchmark, in 3 cases the best optimizer on the RF-based surrogate was not significantly worse than the best one on the real benchmark.

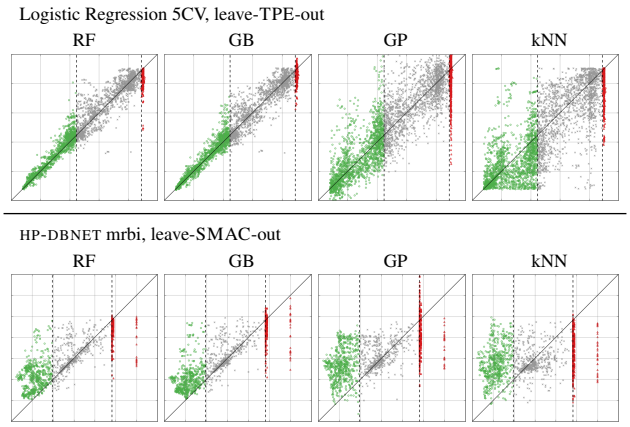

Figure 1: True test performance (x axis) vs. regression model predictions (y axis). We report results for the logistic regression 5CV and HP-DBNET mrbi datasets The models were trained on leave-ooo and tested on the left out data. All plots in one row have the same axes, showing error rates ranging from 0 to 1 (Log.Reg. 5CV) and from 0.4 to 1.1 (HP-DBNET mrbi). Each marker represents the performance of one configuration, vertical lines (and colors) indicate 1/3 best and worst true performance, respectively. Configurations on the diagonal were predicted perfectly, configurations below the diagonal were predicted to perform better than they really did, configurations above the diagonal worse.

Figure 2 studies two benchmarks in more detail, evaluating the performance of various hyperparameter optimizers over time when run on the real benchmark versus the RF-based and GP-based surrogates. The top row shows results on the low-dimensional logistic regression benchmark, a case where

[4] We note that, to enable efficient end-to-end experiments with surrogate benchmarks, the optimizers solely used the surrogate benchmark; no runs of the actual base learner were performed. In order to evaluate the quality of a surrogate model, we could also run the base learner to validate the actual performance of the configurations found when optimizing the surrogate; we report first results for this in the supplemental material.

		Results obtained on real benchmark			Results obtained on RF-based surrogate			Results obtained on GP-based surrogate		
		SMAC	**Spearmint**	**TPE**	**SMAC**	**Spearmint**	**TPE**	**SMAC**	**Spearmint**	**TPE**
Experiment	#evals	Valid. loss	Valid. loss	Valid. loss	Valid. loss	Valid. loss	Valid. loss	Valid. loss	Valid. loss	Valid. loss
Log.Reg.	100	0.08±0.00	**0.07**±0.00	0.08±0.00	0.10±0.02	**0.08**±0.00	0.08±0.01	**-0.06**±0.09	0.07±0.00	0.08±0.04
onlineLDA	50	1266.4±4.4	1264.3±4.9	**1263.7**±3.0	1268.2±2.0	**1265.6**±3.6	1266.1±2.0	1273.0±7.6	**1263.4**±4.5	1268.5±5.3
HP-NNET convex	200	0.19±0.01	0.20±0.01	**0.19**±0.01	**0.21**±0.01	0.21±0.00	0.21±0.01	0.14±0.03	0.14±0.05	**0.12**±0.05
HP-NNET mrbi		0.49±0.01	0.51±0.03	**0.48**±0.01	0.51±0.03	0.54±0.04	**0.49**±0.01	**0.47**±0.02	0.52±0.07	0.47±0.01
HP-DBNET convex		0.15±0.01	0.23±0.10	**0.15**±0.01	**0.18**±0.00	0.20±0.06	0.18±0.00	**0.11**±0.02	0.12±0.05	0.16±0.02
HP-DBNET mrbi		**0.47**±0.02	0.59±0.08	0.47±0.02	0.52±0.02	0.63±0.05	**0.50**±0.01	0.45±0.06	0.62±0.06	**0.41**±0.05
Log.Reg 5CV	500	**0.08**±0.00	0.08±0.00	0.09±0.01	0.08±0.00	**0.08**±0.00	0.09±0.01	**-0.04**±0.07	0.08±0.00	0.07±0.01
HP-NNET convex 5CV		**0.19**±0.01	0.23±0.05	0.21±0.01	**0.22**±0.01	0.24±0.04	0.22±0.01	0.17±0.01	**0.15**±0.05	0.18±0.02
HP-NNET mrbi 5CV		**0.48**±0.01	0.55±0.03	0.51±0.02	**0.51**±0.01	0.59±0.06	0.51±0.02	**0.38**±0.05	0.57±0.06	0.49±0.01

Table 4: Losses obtained for all optimizers and benchmarks. We report results for the real benchmarks (left), RF-based surrogate benchmarks (middle), and GP-based surrogate benchmarks (right), where all surrogate models were learned from leave-ooo data. We report means and standard deviations across 10 runs of each optimizer. For each benchmark, bold face indicates the best mean loss, and underlined values are not statistically significantly different from the best according to an unpaired t-test ($p = 0.05$).

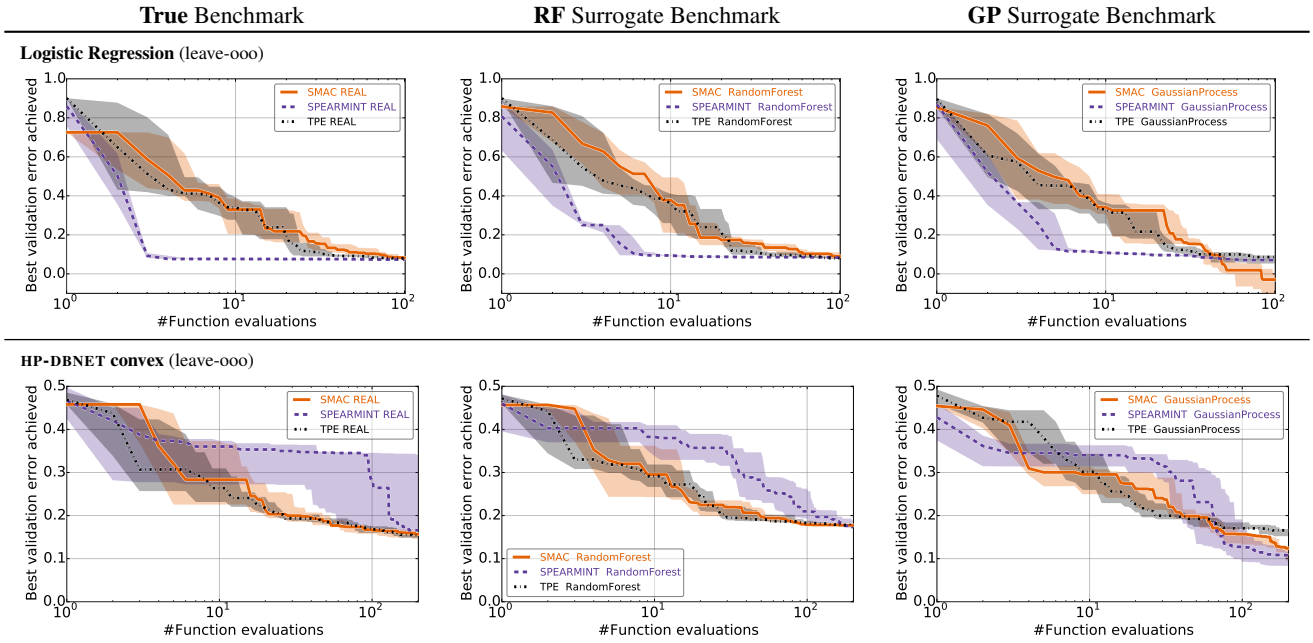

Figure 2: Best performance found by different optimizers over time. We plot median and quartile of best performance across 10 runs of each optimizer over time on the real benchmark (left column) and on surrogates trained on leave-ooo data. Analogous results for all benchmarks are given in Figure B.3 in the supplementary material.

all models performed reasonably well except that the GP surrogate underpredicted the error for some configurations that SMAC found in the end. The second row shows the results for our highest dimensional benchmark: HP-DBNET convex, for which the GP-based surrogate (right figure) yielded qualitatively quite different performance than the real benchmark (left figure). In contrast, the RF-based surrogate yielded performance much closer to that of the real benchmark.

Crucially, the hyperparameter optimization experiments on our surrogate models were much faster than on the real benchmarks. For example, as previously mentioned, a single function evaluation on the real onlineLDA benchmark required up to 10 hours; a surrogate evaluation required less than a second. The hyperparameter optimizer did add a small overhead, leading to costs of 50-60 seconds for a complete hyperparameter optimization experiment on the surrogate benchmarks with TPE or SMAC, as compared to about 340 hours for the real benchmark; a roughly 20 000-fold speedup. Even for SPEARMINT—which added considerably larger overhead than TPE and SMAC due to its MCMC steps—we achieved a 330-fold speedup for an entire hyperparameter optimization experiment.

Conclusion and Future Work

To tackle the high computational cost and overhead of performing hyperparameter optimization benchmarking, we pro-

posed 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 algorithm overhead to a minimum, allowing for 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. Surrogates of low-dimensional benchmarks were almost perfect, while those for high-dimensional benchmarks still yielded acceptable performance. Our final surrogate benchmarks are freely available online at *www.automl.org/benchmarks.html*. Surrogate benchmarks can greatly speed up the development and evaluation of new hyperparameter optimization methods, but we caution that new methods should ultimately be evaluated on (at least some) real benchmarks.

In future work, we intend to study the use of surrogates for general algorithm configuration. Also, hyperparameter optimization and algorithm configuration methods have configuration options themselves, and we hope that good surrogate benchmarks will enable an efficient meta-optimization of these options.

Acknowledgements

This work was supported by the German Research Foundation (DFG) under Emmy Noether grant HU 1900/2-1.

References

Balaprakash, P.; Birattari, M.; and Stützle, T. 2007. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *Hybrid Metaheuristics*. 108–122.

Bardenet, R.; Brendel, M.; Kégl, B.; and Sebag, M. 2013. Collaborative hyperparameter tuning. In *Proc. of ICML'13*.

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

Bergstra, J.; Bardenet, R.; Bengio, Y.; and Kégl, B. 2011. Algorithms for hyper-parameter optimization. In *Proc. of NIPS'11*.

Bergstra, J.; Yamins, D.; and Cox, D. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proc. of ICML'13*, 115–123.

Birattari, M.; Stützle, T.; Paquete, L.; and Varrentrapp, K. 2002. A racing algorithm for configuring metaheuristics. In *Proc. of GECCO'02*, 11–18.

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

Brochu, E.; Cora, V.; and de Freitas, N. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR* abs/1012.2599.

Eggensperger, K.; Feurer, M.; Hutter, F.; Bergstra, J.; Snoek, J.; Hoos, H. H.; and Leyton-Brown, K. 2013. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization*.

Eggensperger, K.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2014. Surrogate benchmarks for hyperparameter optimization. In *ECAI workshop on Metalearning and Algorithm Selection*.

Gorissen, D.; Couckuyt, I.; Demeester, P.; Dhaene, T.; and Crombecq, K. 2010. A surrogate modeling and adaptive sampling toolbox for computer based design. *JMLR* 11:2051–2055.

Guerra, S.; Prudêncio, R.; and Ludermir, T. 2008. Predicting the performance of learning algorithms using support vector machines as meta-regressors. In *Proc. of ICANN'08*, volume 5163, 523–532.

Hoffman, M.; Blei, D.; and Bach, F. 2010. Online learning for latent dirichlet allocation. In *Proc. of NIPS'10*, 856–864.

Hutter, F.; Babić, D.; Hoos, H. H.; and Hu, A. 2007. Boosting verification by automatic tuning of decision procedures. In *Proc. of FMCAD'07*, 27–34.

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

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

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

Komer, B.; Bergstra, J.; and Eliasmith, C. 2014. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. 2012. Imagenet classification with deep convolutional neural networks. In *Proc. of NIPS'12*, 1097–1105.

Larochelle, H.; Erhan, D.; Courville, A.; Bergstra, J.; and Bengio, Y. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proc. of ICML'07*, 473–480.

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

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

Reif, M.; Shafait, F.; Goldstein, M.; Breuel, T.; and Dengel, A. 2014. Automatic classifier selection for non-experts. *PAA* 17(1):83–96.

Sacks, J.; Welch, W.; Welch, T.; and Wynn, H. 1989. Design and analysis of computer experiments. *Statistical Science* 4(4):409–423.

Santner, T.; Williams, B.; and Notz, W. 2003. *The design and analysis of computer experiments*. Springer.

Snoek, J.; Larochelle, H.; and Adams, R. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NIPS'12*, 2960–2968.

Thornton, C.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proc. of KDD'13*, 847–855.