

Efficient benchmarking of algorithm configurators via model-based surrogates

Katharina Eggensperger¹  · Marius Lindauer¹ · Holger H. Hoos² · Frank Hutter¹ · Kevin Leyton-Brown²

Received: 11 May 2016 / Accepted: 4 October 2017 / Published online: 22 December 2017
© The Author(s) 2017

Abstract The optimization of algorithm (hyper-)parameters is crucial for achieving peak performance across a wide range of domains, ranging from deep neural networks to solvers for hard combinatorial problems. However, the proper evaluation of new algorithm configuration (AC) procedures (or *configurators*) is hindered by two key hurdles. First, AC scenarios are hard to set up, including the target algorithm to be optimized and the problem instances to be solved. Second, and even more significantly, they are computationally expensive: a single configurator run involves many costly runs of the target algorithm. Here, we propose a benchmarking approach that uses *surrogate scenarios*, which are computationally cheap while remaining close to the original AC scenarios. These surrogate scenarios approximate the response surface corresponding to true target algorithm performance using a regression model. In our experiments, we construct and evaluate surrogate scenarios for hyperparameter optimization as well as for AC problems that involve performance optimization of solvers for hard combinatorial problems. We generalize previous work by building surrogates for AC scenarios with multiple problem instances, stochastic target algorithms and censored running time observations. We show that our surrogate scenarios capture overall important

Editors: Pavel Brazdil and Christophe Giraud-Carrier.

✉ Katharina Eggensperger
eggensp@cs.uni-freiburg.de

Marius Lindauer
lindauer@cs.uni-freiburg.de

Holger H. Hoos
hoos@cs.ubc.ca

Frank Hutter
fh@cs.uni-freiburg.de

Kevin Leyton-Brown
kevinlb@cs.ubc.ca

¹ University of Freiburg, Freiburg im Breisgau, Germany

² University of British Columbia, Vancouver, BC, Canada

characteristics of the original AC scenarios from which they were derived, while being much easier to use and orders of magnitude cheaper to evaluate.

Keywords Algorithm configuration · Hyper-parameter optimization · Empirical performance model

1 Introduction

The performance of many algorithms (notably, both machine learning procedures and solvers for hard combinatorial problems) depends critically on (hyper-)parameter settings, which are often difficult or costly to optimize. This observation has motivated a growing body of research into automatic methods for finding good settings of such parameters. Recently, sequential model-based Bayesian optimization methods have been shown to outperform more traditional methods for hyperparameter optimization (such as grid search and random search) and to rival or surpass the results achieved by human domain experts (Snoek et al. 2012; Thornton et al. 2013; Bergstra et al. 2014; Lang et al. 2015).

Hyperparameter optimization (HPO) aims to find a hyperparameter setting θ from the space of all possible settings Θ of a given learning algorithm that minimizes expected loss on completely new data (where the expectation is taken over the data distribution). This is often approximated as the blackbox problem of finding a setting that optimizes cross-validation error $\mathcal{L}(\theta)$:

$$\theta^* \in \arg \min_{\theta \in \Theta} \mathcal{L}(\theta). \quad (1)$$

In the more general *algorithm configuration (AC)* problem, the goal is to optimize a performance metric $m : \Theta \times \Pi \rightarrow \mathbb{R}$ of any type of algorithm (the so-called *target algorithm*) across a set of *problem instances* $\pi \in \Pi$, i.e., to find¹

$$\theta^* \in \arg \min_{\theta \in \Theta} \frac{1}{|\Pi|} \sum_{\pi \in \Pi} m(\theta, \pi). \quad (2)$$

The concept of problem instances arises naturally when optimizing the performance of parameterized solvers for combinatorial problems, such as the propositional satisfiability problem (SAT), but we also use this concept to model individual cross-validation folds in hyperparameter optimization (see, e.g., Thornton et al. 2013). HPO is thus a special case of AC.

General-purpose configurators, i.e., algorithm configuration procedures that solve AC problems, such as *ParamILS* (Hutter et al. 2009), *GGA* (Ansótegui et al. 2009, 2015), *irace* (López-Ibáñez et al. 2016) and *SMAC* (Hutter et al. 2011b) have been shown to substantially improve the performance of state-of-the-art algorithms for a wide range of combinatorial problems including SAT (Hutter et al. 2007, 2017), answer set programming (Gebser et al. 2011; Silverthorn et al. 2012), AI planning (Vallati et al. 2013) and mixed integer programming (Hutter et al. 2009), and have also been used to find good instantiations of machine learning frameworks (Thornton et al. 2013; Feurer et al. 2015a) and good architectures and hyperparameters for deep neural networks (Domhan et al. 2015).

¹ We assume, w.l.o.g., that the given performance metric m is to be minimized. Problems of maximizing m' can simply be treated as minimization problems of the form $m = -m'$.

1.1 Obstacles for research on algorithm configuration

One obstacle to further progress in AC is a paucity of reproducible experiments and empirical studies. The hyperparameter optimization library HPOlib (Eggenesperger et al. 2013) and the algorithm configuration library AClib (Hutter et al. 2014a) represent first steps towards alleviating this problem. Each scenario in these libraries consists of a parameterized algorithm and a set of inputs on which the target algorithm should be optimized. These inputs are problem instances for AC scenarios and datasets for HPO scenarios. All scenarios offer a unified interface, making it easier to systematically compare different approaches. However, even with such benchmark libraries available, it is still challenging to assess the performance of configurators in a principled and reproducible manner, for several reasons:

1. A mundane, but often significant, obstacle is *obtaining someone else's implementation* of a target algorithm to work on one's own system. This can involve resolving dependencies, acquiring required software licenses, and obtaining the appropriate input data (which may involve confidentiality agreements or IP restrictions).
2. Some target algorithms require *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 et al. 2012).
3. Running even one configuration of a target algorithm can require minutes or hours, and hence *evaluating hundreds or even thousands of different algorithm configurations* is often quite expensive, requiring days of wall-clock time on a large computer cluster. The computational expense of comprehensive experiments can therefore be prohibitive for research groups lacking access to large-scale computing infrastructure.

1.2 Contributions

In this work, we show that we can use surrogate models to construct cheap-to-evaluate *surrogate AC scenarios* that offer a practical alternative for AC benchmarking experiments by replacing expensive evaluations of the true performance $m(\theta, \pi)$ of a target algorithm configuration θ on a problem instance π with a much cheaper model prediction $\hat{m}(\theta, \pi)$. These surrogate AC scenarios are syntactically equivalent to the original AC scenarios they replace, i.e., sharing the same hyperparameter spaces, instances, etc. (c.f. the formal definition of AC in Eq. 1).

To construct such surrogate scenarios, we leverage *empirical performance models* (EPMs; Leyton-Brown et al. 2009; Hutter et al. 2014b)—regression models that characterize a given algorithm's performance across problem instances and/or parameter settings. The construction of such surrogate models is an expensive offline step,² but once such a model has been obtained, it can be used repeatedly as the basis for efficient experiments with new configurators. Figures 1 and 2 schematically illustrate the workflow for running a configurator on the original scenario and the corresponding surrogate scenario, respectively.

Our surrogate scenarios can be useful in several different ways, including:

1. They can be used to speed up debugging and unit testing of configurators, since our surrogate scenarios are syntactically the same as the original ones (e.g., to test conditional, categorical, and continuous parameters; or to test reasoning across instances). Thus they facilitate the development and effective use of such algorithms.

² By far the most expensive part of this offline step is gathering target algorithm performance data by executing the algorithm with various parameter settings on multiple problem instances. However, as we describe in more detail in Sect. 3 this data can be gathered as a by-product of running configurators on the algorithm.

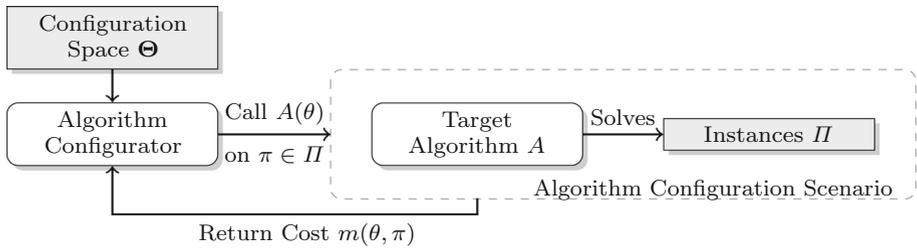


Fig. 1 Workflow of algorithm configuration with target algorithm runs

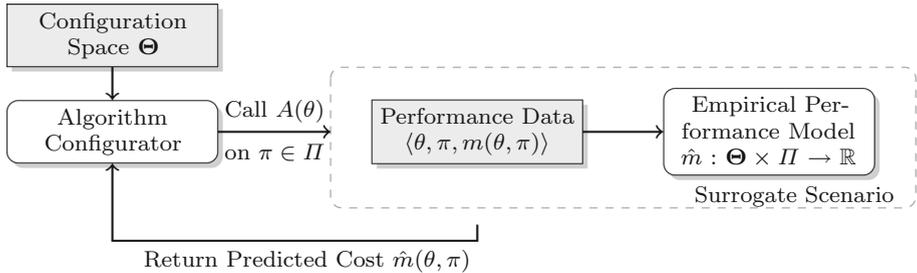


Fig. 2 Workflow of algorithm configuration with surrogate scenario

2. Since the large computational expense of running configurators is typically dominated by the cost of evaluating target algorithm performance under various (hyper-)parameter settings, our scenarios can also substantially reduce the time required for running a configurator, facilitating whitebox testing.
3. Surrogate scenarios that closely resemble original scenarios can also facilitate the evaluation of new features inside a configurator, or even be used for meta-optimization of the parameters of such a procedure. Of course, such meta-optimization can also be performed without using surrogates, albeit at great expense (see, e.g., [Hutter et al. 2009](#)).

This article extends an initial study published at AAI ([Eggenesperger et al. 2015](#)), which focused only on the special case of HPO. Here, we generalize that work to the much more complex general AC problem, handling the problems of optimization across many instances with high-dimensional feature vectors, censored observations due to prematurely-terminated runs, and randomized algorithms.

1.3 Existing work on surrogates

Given the large overhead involved in studying complex scenarios from real-world applications, researchers studying HPO have often fallen back on simple synthetic test functions, such as the Branin function ([Dixon and Szegö 1978](#)), to compare HPO procedures ([Snoek et al. 2012](#)). While such functions are cheap to evaluate, they are not representative of realistic HPO problems because they are smooth and often have very different shapes. Furthermore, they only involve real-valued parameters and hence do not incorporate the categorical and conditional hyperparameters typical of many hyperparameter optimization scenarios.

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. Such a table-based surrogate is trivial to use on a

new system, without any 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 simplify experiments (Birattari et al. 2002; Snoek et al. 2012; Bardenet et al. 2014; Feurer et al. 2015b; Wistuba et al. 2015).

Unfortunately, table lookup is limited to small, finite hyperparameter spaces. Here, we generalize the idea of such surrogates to potentially high-dimensional spaces that may include real-valued, categorical, and conditional hyperparameters. As with table lookup, we first evaluate many hyperparameter configurations in an expensive offline phase. However, 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.

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 used extensively to predict the performance of algorithms across various datasets based on dataset features. Similarly, in the field of algorithm selection (Rice 1976; for a survey see Kotthoff 2014) regression models have been used to predict the performance of algorithms on problem instances (e.g., a satisfiability formula) to select the most promising one (e.g., Nudelman et al. 2003; Xu et al. 2008; Gebser et al. 2011). The statistics literature on the design and analysis of computer experiments (DACE; Sacks et al. 1989; Santner et al. 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 et al. 2010; Hutter et al. 2011b; Shahriari et al. 2016). 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: obtaining high-fidelity *surrogate scenarios as an end in itself, rather than as a means to identifying good points in the space*. In that vein—as previously mentioned—our approach is more similar to work on empirical performance models (Leyton-Brown et al. 2009; Hutter et al. 2014b).

1.4 Structure of the article

The remainder of this article is structured as follows. First, we provide background on the AC problem in Sect. 2, paying particular attention to how it generalizes HPO and on the existing approaches for solving the AC problem we used in our experiments. In Sect. 3, we describe how to use EPMS as surrogates to efficiently benchmark new configurators, introducing the use of quantile regression forests (Meinshausen 2006) for modelling the performance of randomized algorithms. In Sect. 4, we apply our surrogates to application scenarios from ACLib, showing that they model target algorithm performance well enough to yield surrogate AC scenarios which behave qualitatively similarly to the original scenarios, while giving rise to dramatically (up to 1641 times) faster benchmarking experiments.

2 Background on algorithm configuration

In this section, we provide background information on how AC generalizes HPO and hence, how this paper addresses challenges that were not addressed by our previous work

(Eggenesperger et al. 2015). Furthermore, we briefly describe the existing methods for solving AC that we used in our experiments.

2.1 AC as a generalization of HPO

Going beyond the high level description of the AC problem given in Eq. (2), more formally it is defined as follows.

Definition 1 (*Algorithm Configuration*) An instance of the *algorithm configuration problem* is a 6-tuple $(A, \Theta, \mathcal{D}_\Pi, \kappa, \mathcal{F}, m)$ where:

- A is a parameterized target algorithm;
- Θ is the parameter configuration space of A ;
- \mathcal{D}_Π is a distribution over a set of instances Π ;
- $\kappa < \infty$ is a cutoff time at which each run of A will be terminated;
- $\mathcal{F} : \Pi \rightarrow \mathbb{R}^d$ maps each instance to a d -dimensional vector of characteristics that describe the instance. This is an optional input; if no features are available, this is modelled by setting d to 0;
- $m : \Theta \times \Pi \rightarrow \mathbb{R}$ is a function (e.g., running time) that measures the observed cost of running $A(\theta)$ on an instance $\pi \in \Pi$ with cutoff time κ .

The goal is to find

$$\theta^* \in \arg \min_{\theta \in \Theta} \mathbb{E}_{\pi \sim \mathcal{D}_\Pi} (m(\theta, \pi)). \quad (3)$$

Notably, this definition includes a *cutoff time* since in practice we cannot run algorithms for an infinite amount of time, and we need to attribute some (poor) performance value to runs that time out unsuccessfully. We refer to a concrete instantiation of such an AC problem as an AC scenario.

In most AC scenarios (e.g., those in the algorithm configuration library, AClib), \mathcal{D}_Π is chosen as the uniform distribution over a representative set of instances from Π . In practice, we use a set of training instances Π_{Train} from \mathcal{D}_Π to configure our algorithm A and a disjoint set test instances Π_{Test} from \mathcal{D}_Π to evaluate the performance of the configuration finally returned by a configurator, also called the *final incumbent configuration*. Using this training–test split, we can identify over-tuning effects (Hutter et al. 2009), i.e., improvements in performance on Π_{Train} that do not generalize to Π_{Test} . We note that it is typically too expensive to use cross-validation to average over multiple training–test splits, because even single runs of a configurator often require multiple CPU days.

This general AC problem generalizes the HPO problem described by Eq. (1) in various ways.

1. *Types of target algorithms* While HPO only deals with machine learning algorithms, AC deals with arbitrary ones, such as SAT solvers (Hutter et al. 2017) or configurable software systems (Sarkar et al. 2015).
2. *Performance metrics* While HPO typically minimizes one of various loss functions concerning the predictions of a machine learning algorithm, AC includes more general performance metrics, such as running time, approximation error, memory requirements, plan length, or latency.
3. *Randomized algorithms* While the definition of HPO in Eq. (1) concerns deterministic algorithms, the general AC problem includes randomized algorithms. For example, randomized SAT solvers are known to have running time distributions that resemble

exponential distributions (Hoos and Stützle 2004), and it is entirely normal that running times with two different pseudo-random number seeds can differ by an order of magnitude.

4. *Multiple instances* Equation (2) already shows that the goal in the AC problem is to minimize the given performance metric *on average* across instances π from a distribution \mathcal{D}_Π . HPO can also be cast as optimization across cross-validation folds, in which case these are modelled as instances for configurators; these configurators will then evaluate configurations on one fold at a time and only evaluate additional folds for promising configurations.
5. *Prevalence of many categorical and conditional parameters* While the parameter space in current HPO scenarios is often fairly low-dimensional and continuous, general AC scenarios often feature many discrete choices between algorithm components, as well as conditional parameters that only apply to some algorithm components. We note, however, that high-dimensional spaces with categorical parameters and high degrees of conditionality also exist in HPO, e.g., in the optimization of machine learning frameworks (Thornton et al. 2013; Feurer et al. 2015a) or architectural optimization of deep neural networks (Bergstra et al. 2014; Domhan et al. 2015).
6. *Features* In most AC scenarios, each instance is described by a vector of features. Examples of such features range from simple problem size measurements to “probing” or “landmarking” features derived from the behaviour of an algorithm when run on the instance for a bounded amount of time (e.g., number of restarts or constraint propagations in the case of SAT). Instance features are a crucial ingredient in EPMs (Leyton-Brown et al. 2009; Hutter et al. 2014b), which, as mentioned earlier, predict the performance $m(\theta, \pi)$ of a target algorithm configuration θ on a problem instance π . They have been studied even more extensively in the context of the per-instance algorithm selection problem (Nudelman et al. 2003, 2004; Xu et al. 2008; Malitsky et al. 2013; Kotthoff 2014; Lindauer et al. 2015; Bischl et al. 2016), where, given a portfolio of algorithms \mathcal{P} , the goal is to find a mapping $s : \Pi \rightarrow \mathcal{P}$. Thus, feature extractors are available for many problems, including mixed integer programming (MIP; Leyton-Brown et al. 2009; Kadioglu et al. 2010; Xu et al. 2011; Hutter et al. 2014b), propositional satisfiability (SAT; Nudelman et al. 2004; Xu et al. 2008; Hutter et al. 2014b), answer set programming (ASP; Maratea et al. 2014; Hoos et al. 2014), and meta-learning (Gama and Brazdil 1995; Köpf et al. 2000; Bensusan and Kalousis 2001; Guerra et al. 2008; Leite et al. 2012; Reif et al. 2014; Schilling et al. 2015). Recently, Loreggia et al. (2016) proposed the use of deep neural networks to generate instance features automatically, which may be useful when expert-crafted features are unavailable.
7. *Censored observations* For AC scenarios where the goal is to minimize running time, it is common practice for configurators to terminate poorly performing runs early in order to save time. This idea is closely related to the idea of racing (Maron and Moore 1994). This *adaptive capping* process yields performance measurements that only constitute a lower bound to the actual running time and need to be modelled as such.

2.2 Algorithm configurators

While the HPO community has focused quite heavily on the Bayesian optimization framework (Brochu et al. 2010; Shahriari et al. 2016), widely studied approaches for AC are more diverse. These include *ParamILS*, which performs iterated local search (Hutter et al. 2009), *GGA*, which leverages genetic algorithms (Ansótegui et al. 2009, 2015), *irace*, a generalization of racing methods (López-Ibáñez et al. 2016; Lang et al. 2015), *OpenTuner*, which

leverages bandit solvers on top of a set of search heuristics (Ansel et al. 2014), *SMAC*, an approach based on Bayesian optimization (Hutter et al. 2011b), and *ROAR*, the specialization of this method to random sampling without a model. We experimented with all of these methods except *GGA* and *OpenTuner*, the former because in its current version it is not fully compatible with the algorithm configuration scenarios used in our experiments, and the latter because it does not consider problem instances and is therefore not efficiently applicable to our algorithm configuration scenarios, which vary substantially across instances.

We now give more complete descriptions of the state-of-the-art configurators used in our experiments: *ParamILS*, *irace*, *ROAR* and *SMAC*.

ParamILS (Hutter et al. 2009) combines iterated local search (i.e., hill climbing in a discrete neighborhood with perturbation steps and restarts) with methods for efficiently comparing pairs of configurations. Due to its local search approach, *ParamILS* usually compares pairs of configurations that differ in only one parameter value, but occasionally jumps to completely different configurations. When comparing a pair of configurations, assessing performance on all instances is often far too expensive (e.g., requiring solving hundreds of SAT problems). Thus, the *FocusedILS* variant of *ParamILS* we consider here uses two methods to quickly decide which of two configurations is better. First, it employs an *intensification* mechanism to decide how many instances to run for each configuration (starting with a single run and adding runs only for high-performing configurations). Second, it incorporates *adaptive capping*—a technique based on the idea that, when comparing configurations θ_1 and θ_2 with respect to an instance set $\Pi_{sub} \subset \Pi$, evaluation of θ_2 can be terminated prematurely when θ_2 's aggregated performance on Π_{sub} is provably worse than that of θ_1 .

irace (López-Ibáñez et al. 2016) is based on the F-race procedure (Birattari et al. 2002), which aims to quickly decide which of a sampled set of configurations performs significantly best. After an initial race among uniformly sampled configurations, *irace* adapts its sampling distribution according to these results, aiming to focus on promising areas of the configuration space.

ROAR (Hutter et al. 2011b) samples configurations at random and uses an intensification and adaptive capping scheme similar to that of *ParamILS* to determine whether the sampled configuration should be preferred to the current incumbent. As shown by Hutter et al. (2011b), despite its simplicity, *ROAR* performs quite well on some algorithm configuration scenarios.

SMAC (Hutter et al. 2011b) extends Bayesian optimization to handle the more general AC problem. More specifically, it uses previously observed $(\theta, \pi, m(\theta, \pi))$ pairs to learn an EPM to model $p_{\hat{m}}(m | \theta, \pi)$. This EPM is used in a sequential optimization process as follows. After an initialization phase, *SMAC* iterates over the following three steps: (1) use the performance measurements observed so far to fit a marginal random forest model $\hat{f}(\theta) = \mathbb{E}_{\pi \sim \Pi_{train}} [\hat{m}(\theta, \pi)]$; (2) use \hat{f} to select promising configurations $\Theta_{next} \subset \Theta$ to evaluate next, trading off exploration in new parts of the configuration space and exploitation in parts of the space known to perform well by blending optimization of expected improvement with uniform random sampling; and (3) run the configurations in Θ_{next} on one or more instances and compare their performance to the best configuration observed so far. *SMAC* also uses intensification and adaptive capping. However, since adaptive capping leads to right-censored data (i.e., we stop a target algorithm run before reaching the running time cutoff because we already know that it will perform worse than our current incumbent), this data is imputed before being passed to the EPM (Schmee and Hahn 1979; Hutter et al. 2011a).

3 Surrogates of general AC benchmarks

In this section, we show how to construct surrogates of general AC scenarios. In contrast to our earlier work on surrogate scenarios of the special case of HPO (Eggensperger et al. 2015), here we need to take into account the many ways in which AC is more complex than HPO (see Sect. 2.1). In particular, we describe the choices we made to deal with multiple instances and high-dimensional feature spaces; high-dimensional and partially categorical parameter spaces; censored observations; different performance metrics (in particular running time); and randomized algorithms.

3.1 General setup

To construct the surrogate for an AC scenario X , we train an EPM \hat{m} on performance data previously gathered on scenario X (see Sect. 3.2). The surrogate scenario X'_m based on EPM \hat{m} is then structurally identical to the scenario X in all aspects except that it uses predictions instead of measurements of the true performance; in particular, the surrogate's configuration space (including all parameter types and domains) and configuration budget are identical to X . Importantly, the wall clock time to run a configurator on X'_m can be much lower than that required on X , since expensive evaluations in X can be replaced by cheap model evaluations in X'_m .

Our ultimate aim is to ensure that *configurators perform similarly on the surrogate scenario as on the original scenario*. Since effective configurators spend most of their time in high-performance regions of the configuration space, and since relative differences between the performance of configurations in such high-performance regions tend to impact which configuration will ultimately be returned, accuracy in high-performance regions of the space is more important than in regions where performance is poor. Training data should therefore be sampled primarily in high-performance regions. Our preferred way of doing this is to collect performance data primarily via runs of existing configurators. As an additional advantage of this strategy, we can obtain this costly performance data as a by-product of executing configurators on the original scenario.

In addition to gathering data from high-performance regions of the space, it is also important to cover the entire space, including potentially low-performance regions. To get samples that are neither biased towards good nor bad configurations, we 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 parameter spaces.)

1. Inactive parameters were replaced by their default values, or if no default value was specified, by the midpoint of their range of values;³
2. Since our EPMs handle categorical variables natively (see Sect. 3.3), we did not need to encode these. For EPMs that cannot handle categorical variables (e.g., a Gaussian process with a Matérn kernel), we would apply a one-hot-encoding⁴ to the categorical parameter values;
3. We observed that for most algorithms there were parameter combinations that led to crashes (Hutter et al. 2010; Manthey and Lindauer 2016). In our experiments, we only

³ There exist other imputation strategies for missing values (e.g., mean, median, most frequent). In preliminary experiments, we also tried to impute inactive parameters with values outside of their value ranges, but this made no difference in the accuracy of our trained RF-based EPMs.

⁴ One-hot-encoding encodes a categorical variable with k possible values by introducing k binary variables and setting to 1 the binary variable that corresponds to the original variable value, setting all others to zero.

used successful runs (i.e., returning a correct solution within the time budget) and time-outs, as our models cannot classify target algorithm runs into successful and failed runs.⁵

3.2 What kind of data to collect regarding instances?

EPMs for general AC need to make accurate predictions in both the spaces of parameter configurations and of problem instances (in contrast to the special case of HPO that focuses on the configuration space). This opens up another design dimension for gathering training data: which problem instances should we solve in order to gather data for our model? Algorithm configuration scenarios typically come with fixed sets of training and test instances, Π_{Train} and Π_{Test} . In typical applications of EPMs, we only use data from Π_{Train} to build our model and use Π_{Test} to study its generalization performance in the instance space. If our objective, however, is only to construct surrogate scenarios that resemble the original scenarios, then it is never necessary to generalize beyond the instances in the fixed sets Π_{Train} and Π_{Test} ; to see this, recall that a table-based surrogate is the perfect solution for small configuration spaces, despite the obvious fact that it would not generalize. Restricting the data for our EPM to instances from Π_{Train} is therefore an option, but we can expect better performance if we build our model based on instances from both Π_{Train} and Π_{Test} . In order to assess how the choice of instances used by the EPM affects our surrogate scenario, we studied two different setups:

- I *AC and random runs on Π_{Train}* This option only collects data for the EPM on Π_{Train} and relies on the EPM to generalize to Π_{Test} . Specifically, we ran n independent configurator runs on Π_{Train} (in our experiments, n was 10 for each configurator) and also evaluated k runs of randomly sampled (θ, π) pairs with configurations $\theta \in \Theta$ and instances $\pi \in \Pi_{\text{Train}}$ (in our experiments, k was 10 000).
- II *Add incumbents on Π_{Test}* This option includes all the runs from Setting I, but additionally uses some limited data from instances Π_{Test} . Namely, it also evaluates the performance of the configurators' incumbents (i.e., their best parameter configurations over time) on Π_{Test} . This is regularly done for evaluating the performance of configurators over time and thus comes at no extra cost for obtaining data for the EPM.

Details on the datasets used for each of these setups are shown in Table 3. We also tried more expensive setups, such as configuration on $\Pi_{\text{Train}} \cup \Pi_{\text{Test}}$, to achieve better coverage of evaluated configurations $\theta \in \Theta$ on Π_{Test} and not only incumbent configurations with good performance. Preliminary experiments indicated that such more expensive setups did not improve the accuracy of our surrogate scenarios in comparison to the results for Setting II, which we show in Sect. 4.

3.3 Choice of regression models for typical AC parameter spaces

In previous work, Hutter et al. (2014b) and Eggenberger et al. (2015) considered several common regression algorithms for predicting algorithm performance: random forests (RFs; Breimann 2001), Gaussian processes (GPs; Rasmussen and Williams 2006), gradient boosting, support vector regression, k -nearest-neighbours, linear regression, and ridge regression. Overall, the conclusion of those experiments was that RFs and GPs outperformed the other methods for this task. In particular, GPs performed best with few continuous parameters (≤ 10) and few training data points ($\leq 20\,000$); and RFs performed best both when there

⁵ An alternative to removing the crashed runs would be to model them explicitly as unknown constraints (Gelbart et al. 2014).

Table 1 Overview of the hyperparameter ranges used to optimize the RMSE of the random forest and the optimized hyperparameter configuration

Hyperparameter	Ranges	Optimized setting
bootstrapping	{True,False}	False
frac_points	[0.001,1]	0.8
max_nodes	[10, 100,000]	50,000
max_depth	[20, 100]	26
min_samples_in_leaf	[1,20]	1
min_samples_to_split	[2,20]	5
frac_feats	[0.001,1]	0.28
num_trees	[10,50]	48

were many training samples and when parameter spaces were either large or included categorical and continuous parameters.

Since our focus here is on general AC problems that typically involve target algorithms with more than 10 categorical and continuous parameters (see Sect. 4), we limit ourself to RFs in the following. We used our own RF implementation since it natively handles categorical variables. Somewhat surprisingly, in preliminary experiments, we observed that, in our application, the generalization performance of RFs was sensitive to their hyperparameter values. Therefore, we optimized these RF hyperparameters by using *SMAC* across four representative datasets from algorithm configuration (with 5000 subsampled data points and at most 400 function evaluations); the resulting hyperparameter configuration is shown in Table 1. Our RF implementation is publicly available as an open-source project in C++ with a Python interface at https://bitbucket.org/aadfreiburg/random_forest_run.

3.4 Handling widely-varying running times

In AC, a commonly used performance metric is algorithm running time (which is to be minimized). The distribution of running times can strongly vary between different classes of algorithms. In particular algorithms for hard combinatorial problems (e.g. SAT, MIP, ASP) have widely-varying running times across instances. For this reason, we predict log running times $\log(t)$ instead of running times. In log space, the noise is distributed roughly according to a Gaussian, which is the assumption underlying many machine learning algorithms (notably, including RFs minimizing the sum of squared errors).

3.5 Imputation of right-censored data

When considering scenarios in which running time is to be minimized, many configurators use an adaptive capping mechanism to limit algorithm runs to running time κ comparable to the running time of the best seen configuration (see Sect. 2.2). This results in so-called *right-censored data points* for which we only know a lower bound m' on the true performance: $m'(\theta, \pi) \leq m(\theta, \pi)$. Configurators tend to produce many such right-censored data points (in our experiments, 11–38%), and simply discarding these can introduce sizeable bias. We therefore prefer to impute the corresponding running times; as shown by [Hutter et al. \(2011a\)](#), doing so can improve the predictive accuracy of *SMAC*'s EPMs.

Algorithm 1: Imputation of censored data

Input : Uncensored data $\mathbf{X}_u, \mathbf{y}_u$, Censored data $\mathbf{X}_c, \mathbf{y}_c$
Output : Imputed values y_{imp} for \mathbf{X}_c

```

1 EPM.fit( $\mathbf{X}_u, \mathbf{y}_u$ );
2 while not converged do
3   foreach censored sample  $i$  do
4      $\mu, \sigma^2 :=$  EPM.predict( $\mathbf{X}_c^{(i)}$ );
5      $\mathbf{y}_{\text{imp}}^{(i)} :=$  mean of  $\mathcal{N}(\mu, \sigma^2)_{\geq \mathbf{y}_c^{(i)}}$ ;
6   EPM.fit( $\mathbf{X}_u || \mathbf{X}_c, \mathbf{y}_u || \mathbf{y}_{\text{imp}}$ );
7 return  $\mathbf{y}_{\text{imp}}$ 

```

Following Schmee and Hahn (1979) and Hutter et al. (2011a), we use Algorithm 1, which is not specific to RFs, to impute right-censored running time data. We use all uncensored data points, along with their true performance, and all censored data as input. First, we train the EPM on all uncensored data (Line 1). We then compute, for each censored data point, the mean μ and variance σ^2 of the predictive distribution. Since we know a lower bound $\mathbf{y}_c^{(i)}$ on the data point's true running time, we use a truncated normal distribution $\mathcal{N}(\mu, \sigma^2)_{\geq \mathbf{y}_c^{(i)}}$ to update our belief of the true value of $\mathbf{y}^{(i)}$ (Lines 4 and 5). Next, we refit our EPM using the uncensored data and the newly imputed censored data (Line 6). We then iterate this process until the algorithm converges or until 10 iterations have been performed.

3.6 Handling randomized algorithms

Many algorithms are randomized (e.g., RFs or stochastic gradient descent in machine learning, or stochastic local search solvers in SAT solving). In order to properly reflect this in our surrogate scenarios, we should take this randomization into account in our predictions. Earlier work on surrogate scenarios (Eggensperger et al. 2015) only considered deterministic algorithms and only predicted means; when these methods are applied to randomized algorithms, the result is a deterministic surrogate that can differ qualitatively from the original scenario.

Instead, we need to predict the entire distribution $P(Y|X)$ and, when asked to output the performance of a single algorithm run, draw a sample from it. Unfortunately, we do not know in advance the closed form performance distribution—if we knew the running time distributions, we could exploit them in the construction of our EPM (see, e.g., Arbelaez et al. 2016). Instead, to obtain a general solution, we propose to use quantile regression (Koenker 2005; Takeuchi et al. 2006). Following Meinshausen (2006), the α -quantile $Q_\alpha(x)$ is defined by

$$Q_\alpha(x) = \inf\{y : P(Y \leq y|X = x) \geq \alpha\} \quad (4)$$

where the aim is to search for a solution to Eq. (4) such that the probability of $Y \leq y$ is larger than α .

Since we already know that random forests perform well as EPMs (Hutter et al. 2014b; Eggensperger et al. 2015), we use a quantile regression forest (QRF; Meinshausen 2006) for the quantile regression. The QRF is very similar to a RF of regression trees: instead of returning the mean over all labels in the selected leafs of the trees, it returns a given quantile of these labels, $Q_\alpha(x)$. In our surrogate scenarios, when asked to predict a randomized algorithm

performance on $x = \langle \theta, \pi \rangle$ with seed s , we use s to randomly sample a quantile $\alpha \in [0, 1]$ and simply return $Q_\alpha(\theta, \pi)$. For a deterministic algorithm, we return the median $Q_{0.5}(\theta, \pi)$.

4 Experiments for algorithm configuration

Next, we report experimental results for surrogates based on QRFs in the general AC setting. All experiments were performed on Xeon E5-2650 v2 CPUs with 20MB Cache and 64GB RAM running Ubuntu 14.04 LTS.

In the following, we first describe the scenarios we used to evaluate our approach. Then, we report results for the predictive quality of EPMs based on QRFs. Finally, we show that these EPMs are useful as surrogate scenarios, based on an evaluation of the performance of *ParamILS* (Hutter et al. 2009), *ROAR* and *SMAC* (Hutter et al. 2011b), and *irace* (López-Ibáñez et al. 2016) on our surrogate scenarios and the AC scenarios from which our surrogates were derived. For all experiments, we preprocessed the data as described in Sect. 3.1, imputed right-censored data as described in Algorithm 1, and then trained a QRF as described in Sect. 3.6, with the logarithm of the penalized average running time (PAR10) serving as the response variable for running time optimization scenarios.⁶

4.1 Algorithm configuration benchmarks from AClib

For our experiments, we took our scenarios from the algorithm configuration library AClib (Hutter et al. 2014a, see www.aclib.net). Our first set of scenarios involves running time minimization; these consist of different instance sets taken from each of four widely studied combinatorial problems (mixed-integer programming (MIP), propositional satisfiability (SAT), AI planning, and answer set programming (ASP)) and one or more different solvers for each of these problems (*CPLEX*,⁷ *Lingeling* by Biere (2014), *ProbsAT* by Balint and Schöning (2012), *Minisat-HACK-999ED* by Oh (2014), *Clasp* by Gebser et al. (2012), and *lpg* by Gerevini and Serina (2002)). We used the training-test splits defined in AClib. Key characteristics of these scenarios are provided in Table 2, and the underlying AC scenarios are described in detail in Appendix A.

Since AC is a generalization of HPO, we also generated two surrogate HPO scenarios, which allows us to situate our new results in the context of our previous work (Eggenberger et al. 2015). In these new scenarios, we optimized for misclassification rate ($1 - \text{accuracy}$) on 10-fold cross-validation on training data (90% of the data and then validated the model trained on all of the training data with the final parameter configurations on held-out test data (10% of the data). We considered each cross-validation split to be one instance. We used pseudo instance features in these scenarios by simply assigning the i -th split to feature value i and the test data with feature value $k + 1$ (i.e., 11). Inspired by the automated machine learning tool *auto-sklearn* (Feurer et al. 2015a) and available at <https://bitbucket.org/mlindauer/aclib2>, we configured a *SVM* (Cortes and Vapnik 1995) on MNIST⁸ and *xgboost* (Chen and Guestrin 2016) on covtype⁹ (Collobert et al. 2002).

⁶ PAR10 averages all running times, counting each capped run as having taken 10 times the running time cutoff κ (Hutter et al. 2009).

⁷ <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>.

⁸ <http://www.openml.org/d/554>.

⁹ <http://www.openml.org/d/293>.

We ran *SMAC*, *ROAR*, and *ParamILS* ten times for each running time optimization scenario in order to collect performance data in regions that are likely to be explored by one or more configurators. For the HPO scenarios, we additionally ran *irace* ten times.¹⁰ We report properties of the resulting datasets in Table 3. As described in Sect. 3.2, we used two different setups to collect training data for our EPMS. Due to memory limitations on our machines, we used at most 1 million data points to train our EPMS. If we collected more than 1 million points, we subsampled them to 1 million.

4.2 Evaluation of raw model performance

We now report the predictive performance of QRFs as EPMS. In past work (Hutter et al. 2014b), we performed a similar analysis using random forests as EPMS. However, the training data differed substantially from the data we use in this work. In particular, we uniformly sampled sets of target algorithm configurations and problem instances, and then gathered a performance observation for every entry in the Cartesian product of these sets. Here, in contrast, we use configurators to bias training data towards high-performance regions of the given configuration space; this results in a larger number of configurations in our training data, many of which are evaluated only on few instances. The question of whether effective EPMS can be trained using such sparse and biased data has not previously been studied and is an essential requirement for inclusion in our surrogate scenarios.

In Table 4, we show the predictive accuracy of our trained EPMS based on root mean squared error (RMSE; in log space for running time scenarios) to estimate how far our predictions are from true performance values, and Spearman’s rank correlation coefficient (CC; Spearman 1904) to assess whether we can accurately rank different configurations based on predicted performance values. The latter metric is particularly useful in the context of surrogate scenarios, because a configurator can make correct decisions as long as the *ranking* of configurations is correct, i.e., the EPM predicts poorly performing configurations to be bad and strong configurations to be good.

To obtain an unbiased estimate of generalization performance, we used a leave-one-run-out validation splitting scheme: in each split, we used all but one run of each configurator as training data and evaluated the EPM trained on this data on the remaining runs. All configurators are randomized, and we independently initialized each configurator run with a different random seed. Therefore, all data points evaluated by a single AC run are independent of the points of a different AC run, even though the two runs may contain identical data points.

Table 4 shows our results on held out data, specifically all data collected while configuring the target algorithm on Π_{Train} as well as on all data collected during the validation of the incumbent configurations on Π_{Test} . As expected, the predictive performance of our EPM on Π_{Train} is quite similar between Setting I and II. However on Π_{Test} , Setting II performed significantly better than Setting I across our scenarios (p -values of 0.0021 on RMSE and 0.0067 on CC based on a one-sided, non-parametric permutation test, cf. Hoos (2017)). Overall, our EPMS yielded rather accurate target algorithm performance predictions, and achieved high overall correlation ($CC \geq 0.75$) in 9 out of 11 scenarios wrt Π_{Train} and in all scenarios wrt Π_{Test} using Setting II. The RMSE on the running time scenarios was substantially smaller than 1.0, i.e., the predictions were less than one order of magnitude off on average. Considering the HPO scenarios, our models were more accurate for *svm-mnist* than they were for *xgboost-covertime*. This difference was driven by timeouts (counted using the maximal error value of 1). For *svm-mnist*, these timeouts were easier to predict (mostly

¹⁰ Since *irace* (2.4) does not implement an adaptive capping mechanism, its authors recommend that it should not be used for running time minimization.

Table 2 Properties of our AC scenarios

	# Parameter ca./int./co.(cond.)	# Instance features	Total	# Instances train/test	Conf budget	Cutoff κ
<i>CPLEX-Regions</i>	63/7/4 (4)	148	222	1000/1000	2	10,000
<i>CPLEX-RCW2</i>	63/7/4 (4)	148	222	495/495	5	10,000
<i>Clasp-Rooks</i>	38/30/7 (55)	119	194	484/351	2	300
<i>Lingeling-CF</i>	137/185/0 (0)	119	441	299/302	2	300
<i>ProbSAT-7SAT</i>	5/1/3 (5)	138	128	250/250	2	300
<i>MiniSATHack-K3</i>	10/0/0 (0)	119	129	300/250	2	300
<i>LPG-Satellite</i>	48/5/14 (22)	305	372	2000/2000	2	300
<i>LPG-Zenotravel</i>	48/5/14 (22)	305	372	2000/2000	2	300
<i>Clasp-WS</i>	61/30/7 (63)	38	136	240/240	4	900
<i>svm-mnist</i>	2/1/3 (2)	1	7	10/1	500	1000
<i>xgboost-covertype</i>	0/2/9 (0)	1	12	10/1	500	1000

We report the size of the configuration spaces Θ for the different kinds of parameters (i.e., categorical, integer-valued, continuous and conditionals), the number of instance features, the total number of input features for our EPMs (#parameters + #features), the number of training and test instances, the configuration budget for each configurator run (in days for combinatorial problems and number of function evaluations for HPO scenarios) and the running time cutoff for each target algorithm run (in seconds)

Table 3 Properties of our datasets

	I			II			
	#data 1000	%cen	%to	#data 1000	%cen	%to	#conf 1000
<i>CPLEX-Regions</i>	656	27	0	825	22	< 1	198
<i>CPLEX-RCW2</i>	166	38	2	217	29	1	80
<i>Clasp-Rooks</i>	245	14	3	310	11	5	63
<i>Lingeling-CF</i>	149	21	9	176	17	9	66
<i>ProbSAT</i>	199	17	3	245	14	3	28
<i>MiniSATHack-K3</i>	177	16	2	218	13	2	37
<i>LPG-Satellite</i>	565	27	< 1	969	16	< 1	252
<i>LPG-Zenotavel</i>	685	26	< 1	> 1K	17	1	204
<i>Clasp-WS</i>	172	17	4	207	14	4	56
<i>svm-mnist</i>	29	–	53	29	–	52	19
<i>xgboost-covertime</i>	30	–	4	30	–	2	22

We list the rounded number/1000 of collected (θ, π) pairs of 10 runs of each configurator and the ratio of right-censored runs (#cen) and timeouts (#to) for each of our settings: I, II. We also report the total number/1000 of different configurations (#conf)

driven by a small number of parameters); in contrast, a potential timeout could depend on more complex interactions of parameters in the case of *xgboost-covertime*. The predictions for non-timeout runs for *xgboost-covertime* were roughly as good as for *svm-mnist*. Since Setting II performed consistently better than Setting I, in the following we consider only Setting II.¹¹

4.3 Qualitative evaluation of surrogate scenarios

We now turn to the most important experimental question: how well our QRF-based EPMs work as surrogate scenarios for algorithm configurators. For these experiments, we trained and saved a QRF model on the imputed data from our Setting II (described above). To evaluate these EPMs as AC scenarios, we re-ran our configuration experiments, now obtaining running time measurements from an EPM (running as a background process) rather than the real target algorithm. Doing so reduced the average CPU time required for evaluating a configuration on a single instance from 27.29 ± 100.26 ($\mu \pm \sigma$) to 0.23 ± 0.13 s.

We also considered *leave-one-configurator-out (LOCO)* surrogate scenarios. To construct these, we trained the EPM on data gathered by all but one configurator, and then ran the remaining configurator on this surrogate scenario. With this, we simulate benchmarking a new configurator without having any prior knowledge about the behaviour of this configurator.

¹¹ To study whether it is necessary for our model to distinguish different instances, we also considered another baseline, inspired by a metric used by Soares and Brazdil (2004): We computed the rank correlation between the true running times and the mean running times per configuration across instances (aka mean regressor). A low correlation coefficient indicates that the instances differ in hardness or that the rank of configurations changes between instances. Indeed, for most of our scenarios we obtained a low correlation coefficient $CC \leq 0.35$, indicating that it was necessary for the model to consider instances to obtain accurate predictions. For *svm-mnist*, *xgboost-covertime*, *LPG-Satellite*, and *LPG-Zenotavel*, we obtained $CC \geq 0.79$ for data points used during validation, showing that the instances in these scenarios are rather similar (We note that these numbers are based on observed runs and not for predicting performance on unseen instances or configurations).

Table 4 Leave-one-run-out model performance

	RMSE				CC			
	Configuration		Validation		Configuration		Validation	
	I	II	I	II	I	II	I	II
<i>CPLEX-Regions</i>	0.2	0.19	0.33	0.2	0.92	0.92	0.67	0.9
<i>CPLEX-RCW2</i>	0.12	0.12	0.08	0.08	0.98	0.98	0.99	0.99
<i>Clasp-Rooks</i>	0.35	0.35	0.49	0.42	0.98	0.98	0.98	0.98
<i>Lingeling-CF</i>	0.35	0.35	0.72	0.31	0.86	0.86	0.7	0.93
<i>ProbSAT-7SAT</i>	0.6	0.6	0.82	0.54	0.69	0.69	0.36	0.78
<i>MiniSATHack-K3</i>	0.27	0.27	0.48	0.24	0.96	0.96	0.89	0.97
<i>LPG-Satellite</i>	0.1	0.1	0.14	0.13	0.8	0.8	0.92	0.92
<i>LPG-Zenotravel</i>	0.27	0.29	0.38	0.39	0.7	0.69	0.78	0.77
<i>Clasp-WS</i>	0.31	0.31	0.64	0.42	0.95	0.95	0.85	0.95
<i>svm-mnist</i>	0.06	0.06	0.06	0.02	0.99	0.99	0.7	0.75
<i>xgboost-covertype</i>	0.25	0.26	0.17	0.14	0.87	0.85	0.85	0.85

We report mean root mean squared error (RMSE) and Spearman's rank correlation coefficient (CC) between true and predicted loss (for HPO scenarios; last two rows) and log PAR10 running times (for AC scenarios; all other rows), and across ten runs for which the EPM was trained using data from setting I or II (see Sect. 4.1). For each run, we trained an EPM on all but one configuration run for each considered configurator and report average results across left-out runs. Using the QRF, we predicted the median. We report results on all data collected during configuring the target algorithm on π_{Train} and the data collected during the validation of the incumbent configurations on π_{Test}

When used in the context of AC scenarios, the absolute quality of running time predictions is less important than the ranking of the configurators. Therefore we studied performance as a function of time to visually compare the behaviour of different configurators on the original and surrogate scenarios; see Fig. 3. We observe that the relative rankings between configurators were well preserved for surrogate scenarios using EPMs trained on all data. *SMAC* was correctly predicted to outperform *ROAR* in all AC scenarios and at almost all time steps. *ParamILS*'s performance was predicted slightly less well, but ranks were still preserved well across scenarios and time steps. Also, the EPM-based surrogate scenarios captured overtuning effects as present in *LPG-Zenotravel* and *CPLEX-RCW2*. For the machine learning scenarios, we obtained almost perfect surrogate scenarios, with *irace* and *ParamILS* performing similarly, although *ParamILS* achieved a slightly higher inter-quartile ratio on the surrogate than on the original scenario.

In the LOCO setting (Fig. 3, right column), the relative performance of *SMAC* and *ROAR* was still predicted correctly throughout except for the *LPG-Zenotravel* scenario, where *SMAC* and *ROAR* did not improve as much as on the original scenario. *ParamILS*, again, was predicted slightly worse, but its relative ranks were still predicted correctly, with two exceptions. First, on the LOCO surrogate of scenario *CPLEX-RCW2*, *ParamILS* performed worse than on the original scenario (and could not find a configuration better than the default). Second, on the LOCO surrogate of scenario *Clasp-WS*, *ParamILS* performed better than on the original scenario. We believe that our worse results for *ParamILS* were due to the substantial differences in search strategies between *ParamILS* and the other configurators, leading to qualitatively different sets of performance data and hence EPMs; surrogate scenarios constructed based on data from global search algorithms intuitively capture areas of weak/strong performance

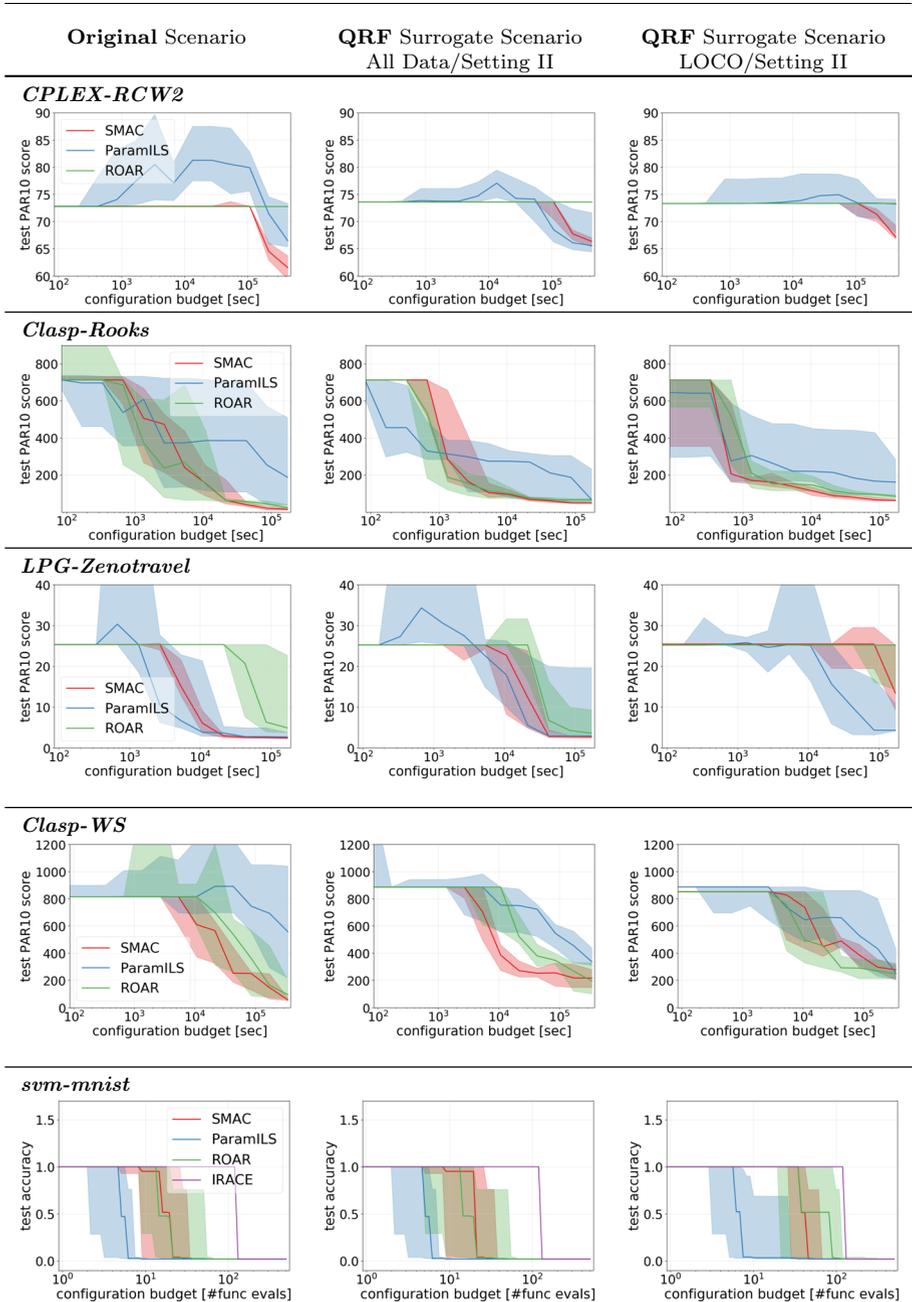


Fig. 3 Best performance of a configuration of the target algorithm (y-axis) found by different configurators versus configuration budget (x-axis). We plot median and quartile of best performance across 10 runs of each configurator on the original scenario (left) and on QRF-based surrogate scenarios trained either on data from all configurators (middle) or leave-one-configurator-out data (right). Please note that for our AC surrogate scenarios *configuration budget [sec]* does not refer to the actual running time, but to predicted running times. The surrogate scenarios needed less time to run, as shown in Table 6

Table 5 Overview of our error metric quantifying the degree to which using surrogates in performance comparisons between two given configurators yielded results statistically significantly different from those obtained based on the underlying original scenarios

Original\surrogate	Better	Equal	Worse
Better	0	0.5	1
Equal	0.5	0	0.5
Worse	1	0.5	0

well, but do not necessarily capture fine local variation and may thus (as discussed above) not work as well for gradient-following configurators.

4.4 Quantitative evaluation of surrogate scenarios

To also provide a quantitative evaluation of how closely our surrogate scenarios resembled the original scenarios, we used an error metric based on the idea that a surrogate scenario should preserve the outcomes of pairwise comparisons of configurators obtained on the underlying original scenarios, across different overall running time budgets. To deal with performance variability due to randomization in target algorithm and configurator runs, we applied statistical tests to determine whether one configurator performed significantly better than another. For each running time budget (number of target algorithm or surrogate evaluations, resp.) starting from κ (or 2), we used a Kruskal–Wallis test and a pairwise post-hoc Wilcoxon rank-sum test with Bonferroni’s multiple testing correction ($\alpha = 0.05$). Table 5 shows how our metric penalizes differences in the outcomes of this statistical test between the original and the surrogate versions of a scenario. This metric was inspired by [Leite and Brazdil \(2010\)](#), but we additionally penalized the case in which true performance values do not differ significantly while our surrogate-based predictions do. To obtain our overall error values, we averaged across the values of this metric for each pair of configurators in our comparison and then averaged the error values thus obtained over the different time budgets considered. We note that this metric provides a quantitative measure of the similarity of the qualitative trajectory diagrams shown in [Fig. 3](#).

In [Table 6](#), we report this metric for both surrogates based on *all data* and for the LOCO setting. All our surrogate scenarios achieved an error lower than 0.5, which indicates that, on average, using our surrogates produces behaviour qualitatively similar to that observed for the underlying target algorithms. In most LOCO experiments, we observed slightly higher error values (albeit still below 0.5), because our EPMs had never seen data from the configurator that was run on the respective surrogate scenario. On the *ProbSAT* scenario, the configurators running on surrogates exhibited qualitatively similar behaviour as on the original scenario and achieved an error of 0, although the EPMs were relatively weak (comparatively high RMSE and low CC; see [Table 4](#)). This occurred because *ProbSAT* only has a few important parameters, which all configurators identified on the original and surrogate scenarios. For *CPLEX-Regions*, [Table 6](#) reports the highest difference between the error on all data and the LOCO setting. In our experiments we observed that for this scenario, *SMAC* achieved similar performance to *ROAR* in the LOCO setting, whereas it performed substantially better on the original scenario. This resulted in a high value of our metric. We observed a similar phenomenon on *xgboost-covertype*, where *ParamILS* found a significantly better-performing configuration earlier and therefore constantly added to the error. Overall, our results indicate

Table 6 The weighted average difference between configurator performance on the original and surrogate scenarios, averaged over time (left; see the text for details on the error metric) and mean running time of individual real target algorithm runs (in CPU seconds), along with the speedup gained when using surrogate scenarios (right). Single predictions in our surrogate-based scenarios required 0.23 ± 0.13 s

Scenario	All data	LOCO	Mean running time	Speed up
<i>CPLEX-Regions</i>	0.17	0.47	7.3	36
<i>CPLEX-RCW2</i>	0.08	0.25	82.1	497
<i>Clasp-Rooks</i>	0.07	0.05	21.3	128
<i>Lingeling-CF</i>	0.1	0.12	36	199
<i>ProbSAT-7SAT</i>	0	0.17	26.5	159
<i>MiniSATHack-K3</i>	0.02	0	30.1	183
<i>LPG-Satellite</i>	0	0.05	8.21	34
<i>LPG-Zenotravel</i>	0.1	0.12	6.79	22
<i>Clasp-WS</i>	0.07	0.15	61.4	383
<i>svm-mnist</i>	0.01	0.05	658	1641
<i>xgboost-covertype</i>	0.18	0.21	566	1434

that the relative performance of configurators on surrogate-based scenarios largely resembled that observed on scenarios involving much costlier target algorithm runs, but that the resemblance was greater when our surrogates were trained based on all available data.

In Table 6, we also report average speedups gained per target algorithm run. Our surrogate scenarios allow dramatic speedups in experimentation, cutting down the time required for the evaluation of individual configurations by a factor of over 1000 for the most expensive AC scenarios.¹² This will substantially ease the development of configurators by facilitating unit testing, debugging, and whitebox testing. Furthermore, the behaviour of configurators on standard scenarios involving actual target algorithm runs was captured closely enough by our surrogate-based scenarios that it makes sense to use the latter in comparative performance evaluations of configurators.

5 Conclusion

We presented a novel approach for constructing model-based surrogate scenarios for the general problem of AC—subsuming HPO. Our surrogate scenarios replace expensive evaluations of algorithm configurations by cheap performance predictions based on EPMs, with orders-of-magnitude speedups for individual evaluations. Our efficient surrogate scenarios can (i) substantially speed up debugging and unit testing of configurators, (ii) facilitate white-box testing, and (iii) provide a basis for assessing and comparing configurator performance.

To construct EPMs for using them as surrogates in AC scenarios, we proposed the use of configurators for generating training data for EPMs, thereby focusing on the more relevant high-performance regions of the parameter configuration space. We further addressed challenges inherent to the AC problem by studying different ways to collect data on training and test instances, imputation of right-censored data, and predicting the performance of ran-

¹² We note, however, that this does *not* imply that entire configurator runs are sped up by the same factor: any overheads of the configurator itself (e.g., in the case of *SMAC* for model building and the internal optimization of its acquisition function) also occur for the surrogate-based scenarios and reduce speedups accordingly.

domized algorithms via novel EPMs based on quantile regression forests. In comprehensive experiments with scenarios from AClib, we showed that our surrogate scenarios were well able to stand in for AC scenarios.

While in this work, we focused on either predicting running time for AC scenarios or loss for HPO scenarios, in general, EPMs could also be trained to predict multiple objectives, such as running time and loss. We defer constructing such more complex surrogate scenarios to future work.

An issue arises from large amounts of target algorithm performance data; for some of our AC scenarios, we had over 1 million data points available, which we subsequently had to subsample to avoid memory issues in the construction of EPMs; however, better solutions to this problem can likely be devised. Since deep neural networks have recently shown impressive results for big data sets and natively support training in batches, we plan to study scalable Bayesian neural networks (Neal 1995; Blundell et al. 2015; Snoek et al. 2015; Springenberg et al. 2016) to predict the performance of randomized algorithms.

Acknowledgements We thank Stefan Falkner for the implementation of the quantile regression forest used in our experiments and for fruitful discussions on early drafts of the paper. K. Eggenesperger, M. Lindauer and F. Hutter acknowledge funding by the DFG (German Research Foundation) under Emmy Noether Grant HU 1900/2-1; K. Eggenesperger also acknowledges funding by the State Graduate Funding Program of Baden-Württemberg. H. Hoos and K. Leyton-Brown acknowledge funding through NSERC Discovery Grants; K. Leyton-Brown also acknowledges funding from an NSERC E.W.R. Steacie Fellowship.

Appendix A: Scenario descriptions

- CPLEX-Regions* is a MIP scenario based on the well-known IBM ILOG *CPLEX* solver, applied to MIP-encoded instances of the combinatorial auction winner determination problem (Leyton-Brown et al. 2000). The MIP instance features used in this scenario include static (Leyton-Brown et al. 2009; Kadioglu et al. 2010; Hutter et al. 2014b) and probing features (Xu et al. 2011). Even though *CPLEX* has 74 parameters, its performance can be predicted quite accurately (Hutter et al. 2014b).
- CPLEX-RCW2* also uses *CPLEX*, in this case to solve MIP-encoded problems from computational sustainability that model habitat preservation for endangered red-cockaded woodpeckers (Ahmadizadeh et al. 2010; Xu et al. 2011). The configuration space and the instance features are the same as in *CPLEX-Regions*; however, *CPLEX* exhibits a much larger range of performance values across *RCW2* instances.
- Clasp-Rooks* is a scenario from the 2014 Configurable SAT Solver Challenge (CSSC'14; Hutter et al. 2017) and is based on the SAT (and ASP) solver *Clasp* (Gebser et al. 2012) applied to so-called “rooks” instances—a variant of the n -queens problem with additional constraints (Manthey and Steinke 2014). We use the instance features generated by the well-known algorithm selector *Satzilla* (Nudelman et al. 2003; Xu et al. 2008; Hutter et al. 2014b) for this and all other SAT scenarios. This AC scenario is distinguished by *Clasp*'s highly structured configuration space, which contains a large number of conditional parameters.
- Lingeling-CF* is also a scenario from CSSC'14; it is based on the state-of-the-art SAT solver *Lingeling* (Biere 2013) applied to circuit-based fuzz testing instances (Brummayer et al. 2012). With 322 parameters, *Lingeling* has

- the largest configuration space of any target algorithm considered in our experiments (and also one of the largest of any SAT solver we are aware of). This gives rise to a particularly challenging AC scenario, because many parameters range from 0 to the maximal 32-bit integer and offer more scope for reductions than improvements in performance.
- ProbSAT-7SAT* is another scenario from CSSC'14; it is based on one of the state-of-the-art local search SAT solvers, *ProbSAT* (Balint and Schönig 2012) applied to 7SAT random instances. With only 9 parameters, the configuration space is quite small.
- MiniSATHack-K3* is our last scenario from the CSSC'14; it is based on a modification of the well-known *MiniSAT* solver (Eén and Sörensson 2003), called *Minisat-HACK-999ED* (Oh 2014) on 3SAT random instances. The 10 categorical parameters give raise to a configuration space of 800 000 parameter configurations, making it the smallest configuration space we consider.
- LPG-Satellite* was introduced in the context of parameter importance analysis with ablation (Fawcett and Hoos 2016). It is based on the AI planning system *lpg* (Gerevini and Serina 2002), which exposes 67 parameters. In this case we study *satellite* instances: planning problems arising in the control and observation scheduling of orbital satellites (Long and Fox 2003). The instance features are a combination of native planning features and features derived by translating planning instances into SAT (Fawcett et al. 2014).
- LPG-Zenotrail* uses the same target algorithm, *lpg*, as *LPG-Satellite*, in combination with instances from the *zenotrail* planning domain (Penberthy and Weld 1994), which arise in a version of route planning. The default configuration of *lpg* achieves worse performance than on *LPG-Satellite*; nevertheless, after configuration, the instances from this scenario turn out to be easier for *lpg*.
- Clasp-WS* is based on the dual-purpose SAT/ASP solver *Clasp* applied to ASP (rather than SAT) instances. *Clasp* has a richer configuration space in the ASP domain (35 additional parameters, including 12 conditional ones); to compensate, we set the configuration budget twice as high as for *Clasp-Rooks*. The ASP problem instances encode optimizing join order in database systems (Lierler and Schüller 2012). To generate instance features, we used the same feature extractor, *claspfe*, as the state-of-the-art ASP algorithm selector *claspfolio* (Hoos et al. 2014).
- svm-mnist* is based on a support vector machine (Cortes and Vapnik 1995) (using the *libsvm* implementation via *scikit-learn*) applied to the well-known MNIST data set. We optimized 6 hyperparameters of the SVM, including the kernel (rbf, polynomial or sigmoid) and its dependent hyperparameters.
- xgboost-coverttype* is based on *xgboost* (Chen and Guestrin 2016) applied to the coverttype data set. We optimized 11 mostly continuous hyperparameters.

References

- Ahmadzadeh, K., Dilkina, B., Gomes, C., & Sabharwal, A. (2010). An empirical study of optimization for maximizing diffusion in networks. In D. Cohen (Ed.) *Proceedings of the international conference on principles and practice of constraint programming (CP'10)*, Lecture Notes in Computer Science (Vol. 6308, pp. 514–521). Berlin: Springer.
- Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., & Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In J. Amaral, & J. Torrellas (Eds.) *International conference on parallel architectures and compilation* (pp. 303–316). New York: ACM.
- Ansótegui, C., Sellmann, M., & Tierney, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. In I. Gent (Ed.) *Proceedings of the fifteenth international conference on principles and practice of constraint programming (CP'09)* Lecture Notes in Computer Science (Vol. 5732, pp. 142–157). Berlin: Springer.
- Ansótegui, C., Malitsky, Y., Sellmann, M., & Tierney, K. (2015). Model-based genetic algorithms for algorithm configuration. In Q. Yang, & M. Wooldridge (Eds.) *Proceedings of the 25th international joint conference on artificial intelligence (IJCAI'15)* (pp. 733–739).
- Arbelaez, A., Truchet, C., & O'Sullivan, B. (2016). Learning sequential and parallel runtime distributions for randomized algorithms. In *Proceedings of the international conference on tools with artificial intelligence (ICTAI)*.
- Balint, A., & Schöning, U. (2012). Choosing probability distributions for stochastic local search and the role of make versus break. In A. Cimatti, & R. Sebastiani (Eds.) *Proceedings of the fifteenth international conference on theory and applications of satisfiability testing (SAT'12)*, Lecture Notes in Computer Science (Vol. 7317, pp. 16–29). Berlin: Springer.
- Bardenet, R., Brendel, M., Kégl, B., & Sebag, M. (2014). Collaborative hyperparameter tuning. In S. Dasgupta, & D. McAllester (Eds.) *Proceedings of the 30th international conference on machine learning (ICML'13)* (pp. 199–207). Madison: Omnipress.
- Bensusan, H., & Kalousis, A. (2001). Estimating the predictive accuracy of a classifier. In *Proceedings of the 12th European conference on machine learning (ECML)* (pp. 25–36). Berlin: Springer.
- Bergstra, J., Yamins, D., & Cox, D. (2014). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In S. Dasgupta, & D. McAllester (Eds.) *Proceedings of the 30th international conference on machine learning (ICML'13)* (pp. 115–123). Madison: Omnipress.
- Biere, A. (2013). Lingeling, plingeling and the sat competition 2013. In A. Balint, A. Belov, M. Heule, & M. Järvisalo (Eds.) *Proceedings of SAT competition 2013: Solver and benchmark descriptions* (Vol. B-2013-1, pp. 51–52). Helsinki: University of Helsinki, Department of Computer Science Series of Publications B.
- Biere, A. (2014). Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In A. Belov, D. Diepold, M. Heule, & M. Järvisalo (Eds.) *Proceedings of SAT competition 2014: Solver and benchmark descriptions* (Vol. B-2014-2, pp. 39–40). Helsinki: University of Helsinki, Department of Computer Science Series of Publications B.
- Birattari, M., Stützle, T., Paquete, L., & Varrentapp, K. (2002). A racing algorithm for configuring metaheuristics. In W. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. Potter, A. Schultz, J. Miller, E. Burke, & N. Jonoska (Eds.) *Proceedings of the genetic and evolutionary computation conference (GECCO'02)* (pp. 11–18). Burlington: Morgan Kaufmann Publishers.
- Bischi, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Frechétte, A., et al. (2016). ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237, 41–58.
- Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015). Weight uncertainty in neural network. In F. Bach, & D. Blei (Eds.) *Proceedings of the 32nd international conference on machine learning (ICML'15)* (Vol. 37, pp. 1613–1622). Madison: Omnipress.
- Brazdil, P., Giraud-Carrier, C., Soares, C., & Vilalta, R. (2008). *Metalearning: Applications to data mining* (1st ed.). Berlin: Springer Publishing Company.
- Breimann, L. (2001). Random forests. *Machine Learning Journal*, 45, 5–32.
- Brochu, E., Cora, V., & de Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Computing Research Repository (CoRR) abs/1012.2599.
- Brummayer, R., Lonsing, F., & Biere, A. (2012). Automated testing and debugging of SAT and QBF solvers. In A. Cimatti, & R. Sebastiani (Eds.) *Proceedings of the fifteenth international conference on theory and applications of satisfiability testing (SAT'12)*, Lecture Notes in Computer Science (Vol. 7317, pp. 44–57). Berlin: Springer.

- Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In B. Krishnapuram, M. Shah, A. Smola, C. Aggarwal, D. Shen, & R. Rastogi (Eds.) *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (KDD)* (pp. 785–794). New York: ACM.
- Collobert, R., Bengio, S., & Bengio, Y. (2002). A parallel mixture of svms for very large scale problems. *Neural Computation*, 14(5), 1105–1114.
- Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273–297.
- Dixon, L., & Szegő, G. (1978). The global optimization problem: An introduction. *Towards global optimization*, 2, 1–15.
- Domhan, T., Springenberg, J. T., & Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In Q. Yang, & M. Wooldridge (Eds.) *Proceedings of the 25th international joint conference on artificial intelligence (IJCAI'15)* (pp 3460–3468).
- Eén, N., & Sörensson, N. (2003). An extensible sat-solver. In E. Giunchiglia, & A. Tacchella (Eds.) *Proceedings of the conference on theory and applications of satisfiability testing (SAT)* Lecture Notes in Computer Science (Vol. 2919, pp. 502–518). Berlin: Springer.
- Eggenberger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., & Leyton-Brown, K. (2013). Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian optimization in theory and practice (BayesOpt'13)*.
- Eggenberger, K., Hutter, F., Hoos, H., & Leyton-Brown, K. (2015). Efficient benchmarking of hyperparameter optimizers via surrogates. In B. Bonet, & S. Koenig (Eds.) *Proceedings of the twenty-ninth national conference on artificial intelligence (AAAI'15)* (pp. 1114–1120). AAAI Press.
- Fawcett, C., & Hoos, H. (2016). Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4), 431–458.
- Fawcett, C., Vallati, M., Hutter, F., Hoffmann, J., Hoos, H., & Leyton-Brown, K. (2014). Improved features for runtime prediction of domain-independent planners. In S. Chien, D. Minh, A. Fern, & W. Ruml (Eds.) *Proceedings of the twenty-fourth international conference on automated planning and scheduling (ICAPS-14)*, AAAI.
- Feurer, M., Klein, A., Eggenberger, K., Springenberg, J. T., & Blum, M., Hutter, F. (2015a). Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, & R. Garnett (Eds.) *Proceedings of the 29th international conference on advances in neural information processing systems (NIPS'15)* (pp. 2962–2970).
- Feurer, M., Springenberg, T., & Hutter, F. (2015b). Initializing Bayesian hyperparameter optimization via meta-learning. In B. Bonet, & S. Koenig (Eds.) *Proceedings of the twenty-ninth national conference on artificial intelligence (AAAI'15)* (pp. 1128–1135). AAAI Press.
- Gama, J., & Brazdil, P. (1995). Characterization of classification algorithms. In *Proceedings of the 7th Portuguese conference on artificial intelligence* (pp. 189–200). Berlin: Springer.
- Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M., & Ziller, S. (2011). A portfolio solver for answer set programming: Preliminary report. In J. Delgrande, & W. Faber (Eds.) *Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (LPNMR'11)*, Lecture Notes in Computer Science (Vol. 6645, pp. 352–357). Berlin: Springer.
- Gebser, M., Kaufmann, B., & Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187–188, 52–89.
- Gelbart, M., Snoek, J., & Adams, R. (2014). Bayesian optimization with unknown constraints. In N. Zhang, & J. Tian (Eds.) *Proceedings of the 30th conference on uncertainty in artificial intelligence (UAI'14)*, AUAI Press.
- Gerevini, A., & Serina, I. (2002). LPG: A planner based on local search for planning graphs with action costs. In M. Ghallab, J. Hertzberg, & P. Traverso (Eds.) *Proceedings of the sixth international conference on artificial intelligence* (pp. 13–22). Cambridge: The MIT Press.
- Gorissen, D., Couckuyt, I., Demeester, P., Dhaene, T., & Crombecq, K. (2010). A surrogate modeling and adaptive sampling toolbox for computer based design. *Journal of Machine Learning Research*, 11, 2051–2055.
- Guerra, S., Prudêncio, R., & Ludermir, T. (2008). Predicting the performance of learning algorithms using support vector machines as meta-regressors. In V. Kurkova-Pohlova, & J. Koutnik (Eds.) *International conference on artificial neural networks (ICANN'08)* (vol. 18, pp. 523–532). Berlin: Springer.
- Hoos, H. (2017). *Empirical algorithmics*. Cambridge: Cambridge University Press. to appear.
- Hoos, H., & Stützle, T. (2004). *Stochastic local search: Foundations and applications*. Burlington: Morgan Kaufmann Publishers Inc.
- Hoos, H., Lindauer, M., & Schaub, T. (2014). claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14, 569–585.

- Hutter, F., Babić, D., Hoos, H., & Hu, A. (2007). Boosting verification by automatic tuning of decision procedures. In L. O’Conner (Ed.) *Formal methods in computer aided design (FMCAD’07)* (pp. 27–34). Washington, DC: IEEE Computer Society Press.
- Hutter, F., Hoos, H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36, 267–306.
- Hutter, F., Hoos, H., & Leyton-Brown, K. (2010). Automated configuration of mixed integer programming solvers. In A. Lodi, M. Milano, & P. Toth (Eds.) *Proceedings of the seventh international conference on integration of AI and OR techniques in constraint programming (CPAIOR’10)* Lecture Notes in Computer Science (Vol. 6140, pp. 186–202). Berlin: Springer.
- Hutter, F., Hoos, H., & Leyton-Brown, K. (2011a). Bayesian optimization with censored response data. In *NIPS workshop on Bayesian optimization, sequential experimental design, and bandits (BayesOpt’11)*.
- Hutter, F., Hoos, H., & Leyton-Brown, K. (2011b). Sequential model-based optimization for general algorithm configuration. In C. Coello (Ed.) *Proceedings of the fifth international conference on learning and intelligent optimization (LION’11)* Lecture Notes in Computer Science (Vol. 6683, pp. 507–523). Berlin: Springer.
- Hutter, F., López-Ibáñez, M., Fawcett, C., Lindauer, M., Hoos, H., Leyton-Brown, K., & Stützle, T. (2014a). Aclib: A benchmark library for algorithm configuration. In P. Pardalos, & M. Resende (Eds.) *Proceedings of the eighth international conference on learning and intelligent optimization (LION’14)* Lecture Notes in Computer Science. Berlin: Springer.
- Hutter, F., Xu, L., Hoos, H., & Leyton-Brown, K. (2014b). Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence*, 206, 79–111.
- Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H., & Leyton-Brown, K. (2017). The configurable SAT solver challenge (CSSC). *Artificial Intelligence*, 243, 1–25.
- Kadioglu, S., Malitsky, Y., Sellmann, M., & Tierney, K. (2010). ISAC—instance-specific algorithm configuration. In H. Coelho, R. Studer, & M. Wooldridge (Eds.) *Proceedings of the nineteenth european conference on artificial intelligence (ECAI’10)* (pp. 751–756). Amsterdam: IOS Press.
- Koenker, R. (2005). *Quantile regression*. Cambridge: Cambridge University Press.
- Kotthoff, L. (2014). Algorithm selection for combinatorial search problems: A survey. *AIMagazine*, 35, 48–60.
- Köpf, C., Taylor, C., & Keller, J. (2000). Meta-analysis: From data characterisation for meta-learning to meta-regression. In *Proceedings of the PKDD-00 workshop on data mining, decision support, meta-learning and ILP*.
- Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, & K. Weinberger (Eds.) *Proceedings of the 26th international conference on advances in neural information processing systems (NIPS’12)* (pp. 1097–1105).
- Lang, M., Kotthaus, H., Marwedel, P., Weihs, C., Rahnenführer, J., & Bischl, B. (2015). Automatic model selection for high-dimensional survival analysis. *Journal of Statistical Computation and Simulation*, 85, 62–76.
- Leite, R., & Brazdil, P. (2010). Active testing strategy to predict the best classification algorithm via sampling and metalearning. In H. Coelho, R. Studer, & M. Wooldridge (Eds.) *Proceedings of the nineteenth European conference on artificial intelligence (ECAI’10)* (pp. 309–314). Amsterdam: IOS Press.
- Leite, R., Brazdil, P., & Vanschoren, J. (2012). Selecting classification algorithms with active testing. In P. Perner (Ed.), *Machine learning and data mining in pattern recognition* Lecture Notes in Computer Science (pp. 117–131). Berlin: Springer.
- Leyton-Brown, K., Pearson, M., & Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the international conference on economics and computation* (pp. 66–76).
- Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4), 22.
- Lierler, Y., & Schüller, P. (2012). Parsing combinatorial categorical grammar via planning in answer set programming. In Lecture Notes in Computer Science (Vol. 7265, pp. 436–453). Berlin: Springer.
- Lindauer, M., Hoos, H., Hutter, F., & Schaub, T. (2015). Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53, 745–778.
- Long, D., & Fox, M. (2003). The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research (JAIR)*, 20, 1–59.
- López-Ibáñez, M., Dubois-Lacoste, J., Caceres, L. P., Birattari, M., & Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3, 43–58.
- Loreggia, A., Malitsky, Y., Samulowitz, H., & Saraswat, V. (2016). Deep learning for algorithm portfolios. In D. Schuurmans, & M. Wellman (Eds.) *Proceedings of the thirtieth national conference on artificial intelligence (AAAI’16)* (pp. 1280–1286). AAAI Press.

- Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2013). Algorithm portfolios based on cost-sensitive hierarchical clustering. In F. Rossi (Ed.) *Proceedings of the 23rd international joint conference on artificial intelligence (IJCAI'13)* (pp. 608–614).
- Manthey, N., & Lindauer, M. (2016). Spybug: Automated bug detection in the configuration space of SAT solvers. In *Proceedings of the international conference on theory and applications of satisfiability testing (SAT)* (pp. 554–561).
- Manthey, N., & Steinke, P. (2014). Too many rooks. In A. Belov, D. Diepold, M. Heule, & M. Järvisalo (Eds.) *Proceedings of SAT competition 2014: Solver and benchmark descriptions* (Vol. B-2014-2, pp. 97–98). University of Helsinki, Department of Computer Science Series of Publications B.
- Maratea, M., Pulina, L., & Ricca, F. (2014). A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming*, 14, 841–868.
- Maron, O., & Moore, A. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In *Proceedings of the 6th international conference on advances in neural information processing systems (NIPS'94)* (pp. 59–66). Burlington: Morgan Kaufmann Publishers.
- Meinshausen, N. (2006). Quantile regression forests. *Journal of Machine Learning Research*, 7, 983–999.
- Neal, R. (1995). Bayesian learning for neural networks. PhD thesis, University of Toronto, Toronto, Canada.
- Nudelman, E., Leyton-Brown, K., Andrew, G., Gomes, C., McFadden, J., Selman, B., & Shoham, Y. (2003). Satzilla 0.9. solver description. *International SAT Competition*.
- Nudelman, E., Leyton-Brown, K., Devkar, A., Shoham, Y., & Hoos, H. (2004). Understanding random SAT: Beyond the clauses-to-variables ratio. In *International conference on principles and practice of constraint programming (CP'04)* (pp. 438–452).
- Oh, C. (2014). Minisat hack 999ed, minisat hack 1430ed and swdia5by. In A. Belov, D. Diepold, M. Heule, & M. Järvisalo (Eds.) *Proceedings of SAT competition 2014: solver and benchmark descriptions* (Vol. B-2014-2, p. 46). University of Helsinki, Department of Computer Science Series of Publications B.
- Penberthy, J., & Weld, D. (1994). Temporal planning with continuous change. In B. Hayes-Roth, & R. Korf (Eds.) *Proceedings of the 12th national conference on artificial intelligence* (pp. 1010–1015). Cambridge: The MIT Press.
- Rasmussen, C., & Williams, C. (2006). *Gaussian processes for machine learning*. Cambridge: The MIT Press.
- Reif, M., Shafait, F., Goldstein, M., Breuel, T., & Dengel, A. (2014). Automatic classifier selection for non-experts. *Pattern Analysis and Applications*, 17(1), 83–96.
- Rice, J. (1976). The algorithm selection problem. *Advances in Computers*, 15, 65–118.
- Sacks, J., Welch, W., Welch, T., & Wynn, H. (1989). Design and analysis of computer experiments. *Statistical Science*, 4(4), 409–423.
- Santner, T., Williams, B., & Notz, W. (2003). *The design and analysis of computer experiments*. Berlin: Springer.
- Sarkar, A., Guo, J., Siegmund, N., Apel, S., & Czarnecki, K. (2015). Cost-efficient sampling for performance prediction of configurable systems. In M. Cohen, L. Grunske, & M. Whalen (Eds.) *30th IEEE/ACM International Conference on Automated Software Engineering* (pp. 342–352). IEEE.
- Schilling, N., Wistuba, M., Drumond, L., & Schmidt-Thieme, L. (2015). Hyperparameter optimization with factorized multilayer perceptrons. In *Machine learning and knowledge discovery in databases* (pp. 87–103). Berlin: Springer.
- Schmee, J., & Hahn, G. (1979). A simple method for regression analysis with censored data. *Technometrics*, 21, 417–432.
- Shahriari, B., Swersky, K., Wang, Z., Adams, R., & de Freitas, N. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1), 148–175.
- Silverthorn, B., Lierler, Y., & Schneider, M. (2012). Surviving solver sensitivity: An ASP practitioner's guide. In A. Dovier, & V. Santos Costa (Eds.) *Technical communications of the twenty-eighth international conference on logic programming (ICLP'12), Leibniz international proceedings in informatics (LIPIcs)* (Vol. 17, pp. 164–175).
- Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, & K. Weinberger (Eds.) *Proceedings of the 26th international conference on advances in neural information processing systems (NIPS'12)* (pp. 2960–2968).
- Snoek J, Rippel O, Swersky K, Kiros R, Satish N, Sundaram N, Patwary M, Prabhat, Adams R (2015). Scalable Bayesian optimization using deep neural networks. In F. Bach, & D. Blei (Eds.) *Proceedings of the 32nd international conference on machine learning (ICML'15)* (Vol. 37, pp. 2171–2180). Madison: Omnipress.
- Soares, C., & Brazdil, P. (2004). A meta-learning method to select the kernel width in support vector regression. *Machine Learning Journal*, 54, 195–209.

- Spearman, C. (1904). The proof and measurement of association between two things. *American Journal of Psychology*, 15, 71–101.
- Springenberg, J., Klein, A., Falkner, S., & Hutter, F. (2016). Bayesian optimization with robust Bayesian neural networks. In *Proceedings of the international conference on advances in neural information processing systems (NIPS'16)*.
- Takeuchi, I., Le, Q., Sears, T., & Smola, A. (2006). Nonparametric quantile estimation. *Journal of Machine Learning Research*, 7, 1231–1264.
- Thornton, C., Hutter, F., Hoos, H., & Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In I. Dhillon, Y. Koren, R. Ghani, T. Senator, P. Bradley, R. Parekh, J. He, R. Grossman, & R. Uthurusamy (Eds.) *The 19th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'13)* (pp 847–855). New York: ACM Press
- Vallati, M., Fawcett, C., Gerevini, A., Hoos, H., & Saetti, A. (2013). Automatic generation of efficient domain-optimized planners from generic parametrized planners. In M. Helmert, & G. Röger (Eds.) *Proceedings of the sixth annual symposium on combinatorial search (SOCS'14)*, AAAI Press.
- Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2015). Learning hyperparameter optimization initializations. In *Proceedings of the international conference on data science and advanced analytics (DSAA)* (pp. 1–10). IEEE
- Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32, 565–606.
- Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*.