Algorithm runtime prediction: Methods & evaluation

Frank Hutter *, Lin Xu, Holger H. Hoos, Kevin Leyton-Brown

Department of Computer Science, University of British Columbia, 201-2366 Main Mall, BC V6T 1Z4, Canada

ARTICLE INFO

ABSTRACT

Perhaps surprisingly, it is possible to predict how long an algorithm will take to run on a previously unseen input, using machine learning techniques to build a model of the algorithm's runtime as a function of problem-specific instance features. Such models have important applications to algorithm analysis, portfolio-based algorithm selection, and the automatic configuration of parameterized algorithms. Over the past decade, a wide variety of techniques have been studied for building such models. Here, we describe extensions and improvements of existing models, new families of models, and—perhaps most importantly—a much more thorough treatment of algorithm parameters as model inputs. We also comprehensively describe new and existing features for predicting algorithm runtime for propositional satisfiability (SAT), travelling salesperson (TSP) and mixed integer programming (MIP) problems. We evaluate these innovations through the largest empirical analysis of its kind, comparing to a wide range of runtime modelling techniques from the literature. Our experiments consider 11 algorithms and 35 instance distributions; they also span a very wide range of SAT, MIP, and TSP instances, with the least structured having been generated uniformly at random and the most structured having emerged from real industrial applications. Overall, we demonstrate that our new models yield substantially better runtime predictions than previous approaches in terms of their generalization to new problem instances, to new algorithms from a parameterized space, and to both simultaneously.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

NP-complete problems are ubiquitous in AI. Luckily, while these problems may be hard to solve on worst-case inputs, it is often feasible to solve even large problem instances that arise in practice. Less luckily, state-of-the-art algorithms often exhibit extreme runtime variation across instances from realistic distributions, even when problem size is held constant, and conversely the same instance can take dramatically different amounts of time to solve depending on the algorithm used [31]. There is little theoretical understanding of what causes this variation. Over the past decade, a considerable body of work has shown how to use supervised machine learning methods to build regression models that provide approximate answers to this question based on given algorithm performance data; we survey this work in Section 2. In this article, we refer to such models as *empirical performance models* (*EPMs*).[1] These models are useful in a variety of practical contexts:

---

* Corresponding author.
  E-mail addresses: hutter@cs.ubc.ca (F. Hutter), xulin730@cs.ubc.ca (L. Xu), hoos@cs.ubc.ca (H.H. Hoos), kevinlb@cs.ubc.ca (K. Leyton-Brown).
  [1] In work aiming to gain insights into instance hardness beyond the worst case, we have used the term *empirical hardness model* [75,76,73]. Similar regression models can also be used to predict objectives other than runtime; examples include an algorithm's success probability [45,97], the solution quality an optimization algorithm achieves in a fixed time [96,20,56], approximation ratio of greedy local search [82], or the SAT competition scoring function [119]. We reflect this broadened scope by using the term EPMs, which we understand as an umbrella that includes EHMs.

- **Algorithm selection.** This classic problem of selecting the best from a given set of algorithms on a per-instance basis [95,104] has been successfully addressed by using EPMs to predict the performance of all candidate algorithms and selecting the one predicted to perform best [18,79,26,45,97,119,70].
- **Parameter tuning and algorithm configuration.** EPMs are useful for these problems in at least two ways. First, they can model the performance of a parameterized algorithm dependent on the settings of its parameters; in a sequential model-based optimization process, one alternates between learning an EPM and using it to identify promising settings to evaluate next [65,7,59,55,56]. Second, EPMs can model algorithm performance dependent on both problem instance features and algorithm parameter settings; such models can then be used to select parameter settings with good predicted performance on a per-instance basis [50].
- **Generating hard benchmarks.** An EPM for one or more algorithms can be used to set the parameters of existing benchmark generators in order to create instances that are hard for the algorithms in question [74,76].
- **Gaining insights into instance hardness and algorithm performance.** EPMs can be used to assess which instance features and algorithm parameter values most impact empirical performance. Some models support such assessments directly [96,82]. For other models, generic feature selection methods, such as forward selection, can be used to identify a small number of key model inputs (often fewer than five) that explain algorithm performance almost as well as the whole set of inputs [76,57].

While these applications motivate our work, in the following, we will not discuss them in detail; instead, we focus on the models themselves. The idea of modelling algorithm runtime is no longer new; however, we have made substantial recent progress in making runtime prediction methods more general, scalable and accurate. After a review of past work (Section 2) and of the runtime prediction methods used by this work (Section 3), we describe four new contributions.

1. We describe new, more sophisticated modelling techniques (based on random forests and approximate Gaussian processes) and methods for modelling runtime variation arising from the settings of a large number of (both categorical and continuous) algorithm parameters (Section 4).
2. We introduce new instance features for propositional satisfiability (SAT), travelling salesperson (TSP) and mixed integer programming (MIP) problems—in particular, novel probing features and timing features—yielding comprehensive sets of 138, 121, and 64 features for SAT, MIP, and TSP, respectively (Section 5).
3. To assess the impact of these advances and to determine the current state of the art, we performed what we believe is the most comprehensive evaluation of runtime prediction methods to date. Specifically, we evaluated all methods of which we are aware on performance data for 11 algorithms and 35 instance distributions spanning SAT, TSP and MIP and considering three different problems: predicting runtime on novel instances (Section 6), novel parameter configurations (Section 7), and both novel instances *and* configurations (Section 8).
4. Techniques from the statistical literature on survival analysis offer ways to better handle data from runs that were terminated prematurely. While these techniques were not used in most previous work—leading us to omit them from the comparison above—we show how to leverage them to achieve further improvements to our best-performing model, random forests (Section 9).[2]

2. An overview of related work

Because the problems have been considered by substantially different communities, we separately consider related work on predicting the runtime of parameterless and parameterized algorithms, and applications of these predictions to gain insights into instance hardness and algorithm parameters.

2.1. Related work on predicting runtime of parameterless algorithms

The use of statistical regression methods for runtime prediction has its roots in a range of different communities and dates back at least to the mid-1990s. In the parallel computing literature, Brewer used linear regression models to predict the runtime of different implementations of portable, high-level libraries for multiprocessors, aiming to automatically select the best implementation on a novel architecture [17,18]. In the AI planning literature, Fink [26] used linear regression to predict how the performance of three planning algorithms depends on problem size and used these predictions for deciding which algorithm to run for how long. In the same community, Howe and co-authors [45,97] used linear regression to predict how both a planner's runtime and its probability of success depend on various features of the planning problem; they also applied these predictions to decide, on a per-instance basis, which of a finite set of algorithms should be run in order to optimize a performance objective such as expected runtime. Specifically, they constructed a *portfolio* of planners that ordered algorithms by their expected success probability divided by their expected runtime. In the constraint programming

---

[2] We used early versions of the new modelling techniques described in Section 4, as well as the extensions to censored data described in Section 9 in recent conference and workshop publications on algorithm configuration [59,55,56,54]. This article is the first to comprehensively evaluate the quality of these models.

literature, Leyton-Brown et al. [75,76] studied the winner determination problem in combinatorial auctions and showed that accurate runtime predictions could be made for several different solvers and a wide variety of instance distributions. That work considered a variety of different regression methods (including lasso regression, multivariate adaptive regression splines, and support vector machine regression) but in the end settled on a relatively simpler method: ridge regression with preprocessing to select an appropriate feature subset, a quadratic basis function expansion, and a log-transformation of the response variable. (We formally define this and other regression methods in Section 3.) The problem-independent runtime modelling techniques from that work were subsequently applied to the SAT problem [90], leading to the successful portfolio-based algorithm selection method SATzilla [89,90,117,119]. Most recently, in the machine learning community, Huang et al. [47] applied linear regression techniques to the modelling of algorithms with low-order polynomial runtimes.

Due to the extreme runtime variation often exhibited by algorithms for solving combinatorial problems, it is common practice to terminate unsuccessful runs after they exceed a so-called *captime*. Capped runs only yield a *lower bound* on algorithm runtime, but are typically treated as having succeeded at the captime. Fink [26] was the first to handle such *right-censored* data points more soundly for runtime predictions of AI planning methods and used the resulting predictions to compute captimes that maximize a given utility function. Gagliolo et al. [28,27] made the connection to the statistical literature on survival analysis to handle right-censored data in their work on dynamic algorithm portfolios. Subsequently, similar techniques were used for SATzilla's runtime predictions [117] and in model-based algorithm configuration [54].

Recently, Smith-Miles et al. published a series of papers on learning-based approaches for characterizing instance hardness for a wide variety of hard combinatorial problems [104,108,106,105]. Their work considered a range of tasks, including not only performance prediction, but also clustering, classification into easy and hard instances, as well as visualization. In the context of performance prediction, on which we focus in this article, theirs is the only work known to us to use neural network models. Also recently, Kotthoff et al. [70] compared regression, classification, and ranking algorithms for algorithm selection and showed that this choice matters: poor regression and classification methods yielded worse performance than the single best solver, while good methods yielded better performance.

Several other veins of performance prediction research deserve mention. Haim & Walsh [37] extended linear methods to the problem of making online estimates of SAT solver runtimes. Several researchers have applied supervised classification to select the fastest algorithm for a problem instance [33,29,34,30,120] or to judge whether a particular run of a randomized algorithm would be good or bad [43] (in contrast to our topic of predicting performance directly using a regression model). In the machine learning community, *meta-learning* aims to predict the accuracy of learning algorithms [111]. Meta-level control for anytime algorithms computes estimates of an algorithm's performance in order to decide when to stop it and act on the solution found [38]. Algorithm scheduling in parallel and distributed systems has long relied on low-level performance predictions, for example based on source code analysis [88]. In principle, the methods discussed in this article could also be applied to meta-level control and algorithm scheduling.

Other research has aimed to identify single quantities that correlate with an algorithm's runtime. A famous early example is the clauses-to-variables ratio for uniform-random 3-SAT [19,83]. Earlier still, Knuth showed how to use random probes of a search tree to estimate its size [69]; subsequent work refined this approach [79,68]. We incorporated such predictors as features in our own work and therefore do not evaluate them separately. (We note, however, that we have found Knuth's tree-size estimate to be very useful for predicting runtime in some cases, *e.g.*, for complete SAT solvers on unsatisfiable 3-SAT instances [90].) The literature on search space analysis has proposed a variety of quantities correlated with the runtimes of (mostly) local search algorithms. Prominent examples include fitness distance correlation [66] and autocorrelation length (ACL) [113]. With one exception (ACL for TSP) we have not included such measures in our feature sets, as computing them can be quite expensive.

2.2. Related work on predicting runtime of parameterized algorithms

In principle, it is not particularly harder to predict the runtimes of parameterized algorithms than the runtimes of their parameterless cousins: parameters can be treated as additional inputs to the model (notwithstanding the fact that they describe the algorithm rather than the problem instance, and hence are directly controllable by the experimenter), and a model can be learned in the standard way. In past work, we pursued precisely this approach, using both linear regression models and exact Gaussian processes to model the dependency of runtime on both instance features and algorithm parameter values [50]. However, this direct application of methods designed for parameterless algorithms is effective only for small numbers of continuous-valued parameters (*e.g.*, the experiments in [50] considered only two parameters). Different methods are more appropriate when an algorithm's parameter space becomes very large. In particular, a careful sampling strategy must be used, making it necessary to consider issues raised in the statistics literature on experimental design. Separately, models must be adjusted to deal with *categorical* parameters: parameters with finite, unordered domains (*e.g.*, selecting which of various possible heuristics to use, or activating an optional preprocessing routine).

The experimental design literature uses the term *response surface model* (*RSM*) to refer to a predictor for the output of a process with controllable input parameters that can generalize from observed data to new, unobserved parameter settings (see, *e.g.*, [14,13]). Such RSMs are at the core of sequential model-based optimization methods for blackbox functions [65], which have recently been adapted to applications in automated parameter tuning and algorithm configuration (see, *e.g.*, [7,6,58,59,55]).

Most of the literature on RSMs of algorithm performance has limited its consideration to algorithms running on single problem instances and algorithms only with continuous input parameters. We are aware of a few papers beyond our own that relax these assumptions. Bartz-Beielstein & Markon [8] support categorical algorithm parameters (using regression tree models), and two existing methods consider predictions across both different instances and parameter settings. First, Ridge & Kudenko [96] applied an analysis of variance (ANOVA) approach to detect important parameters, using linear and quadratic models. Second, Chiarandini & Goegebeur [20] noted that in contrast to algorithm parameters, instance characteristics cannot be controlled and should be treated as so-called *random effects*. Their resulting *mixed-effects* models are linear and, like Ridge & Kudenko's ANOVA model, assume Gaussian performance distributions. We note that this normality assumption is much more realistic in the context of predicting solution quality of local search algorithms (the problem addressed in [20]) than in the context of the algorithm runtime prediction problem we tackle here.

2.3. Related work on applications of runtime prediction to gain insights into instance hardness and algorithm parameters

Leyton-Brown and co-authors [75,90,76] employed forward selection with linear regression models to determine small sets of instance features that suffice to yield high-quality predictions, finding that often as little as five to ten features yielded predictions as good as the full feature set. Hutter et al. [57] extended that work to predictions in the joint space of instance features and algorithm parameters, using arbitrary models. Two model-specific approaches for this joint identification of instance features and algorithm parameters are the ANOVA approach of Ridge & Kudenko [96] and the mixed-effects model of Chiarandini & Goegebeur [20] mentioned previously. Other approaches for quantifying parameter importance include an entropy-based measure [85], and visualization methods for interactive parameter exploration [6].

3. Methods used in related work

We now define the different machine learning methods that have been used to predict algorithm runtimes: ridge regression (used by [17,18,75,76,89,90,50,117,119,47]), neural networks (see [107]), Gaussian process regression (see [50]), and regression trees (see [8]). This section provides the basis for the experimental evaluation of different methods in Sections 6, 7, and 8; thus, we also discuss implementation details.

3.1. Preliminaries

We describe a problem instance by a list of m features $\boldsymbol{z} = [z_1, \ldots, z_m]^\mathsf{T}$, drawn from a given *feature space* $\mathcal{F}$. These features must be computable by a piece of problem-specific code (usually provided by a domain expert) that efficiently extracts characteristics for any given problem instance (typically, in low-order polynomial time w.r.t. to the size of the given problem instance). We define the *configuration space* of a parameterized algorithm with k parameters $\theta_1, \ldots, \theta_k$ with respective domains $\Theta_1, \ldots, \Theta_k$ as a subset of the cross-product of parameter domains: $\boldsymbol{\Theta} \subseteq \Theta_1 \times \cdots \times \Theta_k$. The elements of $\boldsymbol{\Theta}$ are complete instantiations of the algorithm's k parameters, and we refer to them as *configurations*. Taken together, the configuration and the feature spaces define the *input space*: $\mathcal{I} = \boldsymbol{\Theta} \times \mathcal{F}$.

Let $\Delta(\mathbb{R})$ denote the space of probability distributions over the real numbers; we will use these real numbers to represent an algorithm performance measure, such as runtime in seconds on some reference machine. (In principle, EPMs can predict any type of performance measure that can be evaluated in single algorithm runs, such as runtime, solution quality, memory usage, energy consumption, or communication overhead.) Given an algorithm $\mathcal{A}$ with configuration space $\boldsymbol{\Theta}$ and a distribution of instances with feature space $\mathcal{F}$, an EPM is a stochastic process $f : \mathcal{I} \mapsto \Delta(\mathbb{R})$ that defines a probability distribution over performance measures for each combination of a parameter configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ of $\mathcal{A}$ and a problem instance with features $\boldsymbol{z} \in \mathcal{F}$. The prediction of an entire distribution allows us to assess the model's *confidence* at a particular input, which is essential, *e.g.*, in model-based algorithm configuration [7,6,58,55]. Nevertheless, since many of the methods we review yield only point-valued runtime predictions, our experimental analysis focuses on the accuracy of mean predicted runtimes. For the models that define a predictive distribution (Gaussian processes and our variant of random forests), we study the accuracy of confidence values separately in the online appendix, with qualitatively similar results as for mean predictions.

To construct an EPM for an algorithm $\mathcal{A}$ with configuration space $\boldsymbol{\Theta}$ on an instance set Π, we run $\mathcal{A}$ on various combinations of configurations $\boldsymbol{\theta}_i \in \boldsymbol{\Theta}$ and instances $\pi_i \in \Pi$, and record the resulting performance values y_i. We record the k-dimensional parameter configuration $\boldsymbol{\theta}_i$ and the m-dimensional feature vector $\boldsymbol{z}_i$ of the instance used in the ith run, and combine them to form a $p = k + m$-dimensional vector of *predictor variables* $\boldsymbol{x}_i = [\boldsymbol{\theta}_i^\mathsf{T}, \boldsymbol{z}_i^\mathsf{T}]^\mathsf{T}$. The training data for our regression models is then simply $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$. We use $\boldsymbol{X}$ to denote the $n \times p$ matrix containing $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]^\mathsf{T}$ (the so-called *design matrix*) and $\boldsymbol{y}$ for the vector of performance values $[y_1, \ldots, y_n]^\mathsf{T}$.

Various transformations can make this data easier to model. In this article, we focus on runtime as a performance measure and use a log-transformation, thus effectively predicting log runtime.[3] In our experience, we have found this transformation to be very important due to the large variation in runtimes for hard combinatorial problems. We also transformed

---

[3] Due to the resolution of our CPU timer, runtimes below 0.01 seconds are measured as 0 seconds. To make $y_i = \log(r_i)$ well defined in these cases, we count them as 0.005 (which, in log space, has the same distance from 0.01 as the next bigger value measurable with our CPU timer, 0.02).

the predictor variables, discarding those input dimensions constant across all training data points and normalizing the remaining ones to have mean 0 and standard deviation 1 (*i.e.*, for each input dimension we subtracted the mean and then divided by the standard deviation).

For some instances, certain feature values can be missing because of timeouts, crashes, or because they are undefined (when preprocessing has already solved an instance). These *missing values* occur relatively rarely, so we use a simple mechanism for handling them. We disregard missing values for the purposes of normalization, and then set them to zero for training our models. This means that missing feature values are effectively assumed to be equal to the mean for the respective distribution and thus to be minimally informative. In some models (ridge regression and neural networks), this mechanism leads us to ignore missing features, since their weight is multiplied by zero.

Most modelling methods discussed in this paper have free hyperparameters that can be set by minimizing some loss function, such as cross-validation error. We point out these hyperparameters, as well as their default setting, when discussing each of the methods. While, to the best of our knowledge, all previous work on runtime prediction has used fixed default hyperparameters, we also experimented with optimizing them for every method in our experiments. For this purpose, we used the gradient-free optimizer DIRECT [64] to minimize 2-fold cross-validated root mean squared error (RMSE) on the training set with a budget of 30 function evaluations. This simple approach is a better alternative than the frequently-used grid search and random search [9].

3.2. Ridge regression

Ridge regression (see, *e.g.*, [12]) is a simple regression method that fits a linear function $f_{\boldsymbol{w}}(\boldsymbol{x})$ of its inputs $\boldsymbol{x}$. Due to its simplicity (both conceptual and computational) and its interpretability, combined with competitive predictive performance in most scenarios we studied, this is the method that has been used most frequently in the past for building EPMs [26,45, 75,76,90,50,115].

Ridge regression works as follows. Let $\boldsymbol{X}$ and $\boldsymbol{y}$ be as defined above, let $\boldsymbol{I_p}$ be the $p \times p$ identity matrix, and let ϵ be a small constant. Then, compute the weight vector

$$\boldsymbol{w} = \left(\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X} + \epsilon \boldsymbol{I_p}\right)^{-1}\boldsymbol{X}^{\mathsf{T}}\boldsymbol{y}.$$

Given a new feature vector, $\boldsymbol{x}_{n+1}$, ridge regression predicts $f_{\boldsymbol{w}}(\boldsymbol{x}_{n+1}) = \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_{n+1}$. Observe that with $\epsilon = 0$, we recover standard linear regression. The effect of $\epsilon > 0$ is to regularize the model by penalizing large coefficients $\boldsymbol{w}$; it is equivalent to a Gaussian prior favouring small coefficients under a Bayesian model (see, *e.g.*, [12]). A beneficial side effect of this regularization is that numerical stability improves in the common case where $\boldsymbol{X}$ is rank deficient, or nearly so. The computational bottleneck in ridge regression with p input dimensions is the inversion of the $p \times p$ matrix $A = \boldsymbol{X}^{\mathsf{T}}\boldsymbol{X} + \epsilon \boldsymbol{I_p}$, which requires time cubic in p.

Algorithm runtime can often be better approximated by a polynomial function than by a linear one, and the same holds for log runtimes. For that reason, it can make sense to perform a basis function expansion to create new features that are products of two or more original features. In light of the resulting increase in the number of features, a quadratic expansion is particularly appealing. Formally, we augment each model input $\boldsymbol{x}_i = [x_{i,1}, \ldots, x_{i,p}]^{\mathsf{T}}$ with pairwise product inputs $x_{i,j} \cdot x_{i,l}$ for $j = 1, \ldots, p$ and $l = j, \ldots, p$.

Even with ridge regularization, the generalization performance of linear regression (and, indeed, many other learning algorithms) can deteriorate when some inputs are uninformative or highly correlated with others; in our experience, it is difficult to construct sets of instance features that do not suffer from these problems. Instead, we reduce the set of input features by performing *feature selection*. Many different methods exist for feature expansion and selection; we review two different ridge regression variants from the recent literature that only differ in these design decisions.[4]

3.2.1. Ridge regression variant RR: Two-phase forward selection [117,119]

For more than half a decade, we used a simple and scalable feature selection method based on forward selection (see *e.g.*, [36]) to build the regression models used by `SATzilla` [117,119]. This iterative method starts with an empty input set, greedily adds one linear input at a time to minimize cross-validation error at each step, and stops when l linear inputs have been selected. It then performs a full quadratic expansion of these l linear features (using the original, unnormalized features, and then normalizing the resulting quadratic features again to have mean zero and standard deviation one). Finally, it carries out another forward selection with the expanded feature set, once more starting with an empty input set and stopping when q features have been selected. The reason for the two-phase approach is scalability: this method prevents us from ever having to perform a full quadratic expansion of our features. (For example, we have often employed over 100

---

[4] We also considered a third ridge regression variant that was originally proposed by Leyton-Brown et al. [76] ("ridge regression with elimination of redundant features", or RR-el for short). Unfortunately, running this method was computationally infeasible, considering the large number of features we consider in this paper, (a) forcing us to approximate the method, and (b) nevertheless preventing us from performing 10-fold cross-validation. Because these hurdles made it impossible to fairly compare RR-el to other methods, we do not discuss RR-el here. However, for completeness, our online appendix includes both a definition of our approximation to RR-el and experimental results showing it to perform worse than ridge regression variant RR in 34/35 cases.

features and a million runtime measurements; in this case, a full quadratic expansion would involve over 5 billion feature values.)

Our implementation reduces the computational complexity of forward selection by exploiting the fact that the inverse matrix $(A')^{-1}$ resulting from including one additional feature can be computed incrementally by two rank-one updates of the previous inverse matrix A^{-1}, requiring quadratic time rather than cubic time [103].

In our experiments, we fixed the number of linear inputs to $l = 30$ in order to keep the result of a full quadratic basis function expansion manageable in size (with 1 million data points, the resulting matrix has $(\binom{30}{2} + 30) \cdot 1\,000\,000$, or about 500 million elements). The maximum number of quadratic terms q and the ridge penalizer ϵ are free parameters of this method; by default, we used $q = 20$ and $\epsilon = 10^{-3}$.

3.2.2. Ridge regression variant SPORE-FoBa: Forward–backward selection [47]

Recently, Huang et al. [47] described a method for predicting algorithm runtime that they called Sparse POlynomial REgression (SPORE), which is based on ridge regression with forward–backward (FoBa) feature selection.[5] Huang et al. concluded that SPORE-FoBa outperforms lasso regression, which is consistent with the comparison to lasso by Leyton-Brown et al. [76]. In contrast to the RR variants above, SPORE-FoBa employs a cubic feature expansion (based on its own normalizations of the original predictor variables). Essentially, it performs a single pass of forward selection, at each step adding a small *set* of terms determined by a forward–backward phase on a feature's candidate set. Specifically, having already selected a set of terms T based on raw features S, SPORE-FoBa loops over all raw features $r \notin S$, constructing a candidate set T_r that consists of all polynomial expansions of $S \cup \{r\}$ that include r with non-zero degree and whose total degree is bounded by 3. For each such candidate set T_r, the forward–backward phase iteratively adds the best term $t \in T \setminus T_r$, if its reduction of root mean squared error (RMSE) exceeds a threshold γ (forward step), and then removes the worst term $t \in T$, if its reduction of RMSE is below 0.5γ (backward step). This phase terminates when no single term $t \in T \setminus T_r$ can be added to reduce RMSE by more than γ. Finally, SPORE-FoBa's outer forward selection loop chooses the set of terms T resulting from the best of its forward–backward phases, and iterates until the number of terms in T reach a prespecified maximum of t_{max} terms. In our experiments, we used the original SPORE-FoBa code; its free parameters are the ridge penalizer ϵ, t_{max}, and γ, with defaults $\epsilon = 10^{-3}$, $t_{max} = 10$, and $\gamma = 0.01$.

3.3. Neural networks

Neural networks are a well-known regression method inspired by information processing in the human brain. The multilayer perceptron (MLP) is a particularly popular type of neural network that organizes single computational units ("neurons") in layers (input, hidden, and output layers), using the outputs of all units in a layer as the inputs of all units in the next layer. Each neuron n_i in the hidden and output layers with k inputs $\mathbf{a_i} = [a_{i,1}, \dots, a_{i,k}]$ has an associated weight term vector $\mathbf{w_i} = [w_{i,1}, \dots, w_{i,k}]$ and a bias term b_i, and computes a function $\mathbf{w_i}^\mathsf{T} \mathbf{a_i} + b_i$. For neurons in the hidden layer, the result of this function is further propagated through a nonlinear activation function $g : \mathbb{R} \to \mathbb{R}$ (which is often chosen to be tanh). Given an input $\mathbf{x} = [x_1, \dots, x_p]$, a network with a single hidden layer of h neurons $n_1, \dots, n_h$ and a single output neuron n_{h+1} then computes output

$$\hat{f}(\mathbf{x}) = \left(\sum_{j=1}^{h} g(\mathbf{w_j}^\mathsf{T}\mathbf{x} + b_j) \cdot w_{h+1,j} \right) + b_{h+1}.$$

The $p \cdot h + h$ weight terms and $h + 1$ bias terms can be combined into a single weight vector $\mathbf{w}$, which can be set to minimize the network's prediction error using any continuous optimization algorithm (*e.g.*, the classic "backpropagation" algorithm performs gradient descent to minimize squared prediction error).

Smith-Miles & van Hemert [107] used an MLP with one hidden layer of 28 neurons to predict the runtime of local search algorithms for solving timetabling instances. They used the proprietary neural network software Neuroshell, but advised us to compare to an off-the-shelf Matlab implementation instead. We thus employed the popular Matlab neural network package NETLAB [84]. NETLAB uses activation function $g = \tanh$ and supports a regularizing prior to keep weights small, minimizing the error metric $\sum_i^N (\hat{f}(\mathbf{x_i}) - y_i)^2 + \alpha \mathbf{w}^\mathsf{T}\mathbf{w}$, where α is a parameter determining the strength of the prior. In our experiments, we used NETLAB's default optimizer (scaled conjugate gradients, SCG) to minimize this error metric, stopping the optimization after the default of 100 SCG steps. Free parameters are the regularization factor α and the number of hidden neurons h; we used NETLAB's default $\alpha = 0.01$ and, like Smith-Miles & van Hemert [107], $h = 28$.

3.4. Gaussian process regression

Stochastic Gaussian processes (GPs) [94] are a popular class of regression models with roots in geostatistics, where they are also called Kriging models [71]. GPs are the dominant modern approach for building response surface models

---

[5] Although this is not obvious from their publication [47], the authors confirmed to us that FoBa uses ridge rather than LASSO regression, and also gave us their original code.

[98,65,99,6]. They were first applied to runtime prediction by Hutter et al. [50], who found them to yield better results than ridge regression, albeit at greater computational expense.

To construct a GP regression model, we first need to select a kernel function $k : \mathcal{I} \times \mathcal{I} \mapsto \mathbb{R}^+$, characterizing the degree of similarity between pairs of elements of the input space $\mathcal{I}$. A variety of kernel functions are possible, but the most common choice for continuous inputs is the squared exponential kernel

$$k_{\text{cont}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\sum_{l=1}^{p} \left(-\lambda_l \cdot (x_{i,l} - x_{j,l})^2 \right) \right), \tag{1}$$

where $\lambda_1, \ldots, \lambda_p$ are kernel parameters. It is based on the idea that correlations decrease with weighted Euclidean distance in the input space (weighing each dimension l by a kernel parameter λ_l). In general, such a kernel defines a prior distribution over the type of functions we expect. This distribution takes the form of a *Gaussian stochastic process*: a collection of random variables such that any finite subset of them has a joint Gaussian distribution. What remains to be specified is the tradeoff between the strength of this prior and fitting observed data, which is set by specifying the *observation noise*. Standard GPs assume normally distributed observation noise with mean zero and variance σ^2, where σ^2, like the kernel parameters λ_l, can be optimized to improve the fit. Combining the prior specified above with the training data $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ yields the *posterior* distribution at a new input point $\boldsymbol{x}_{n+1}$ (see the book by Rasmussen & Williams [94] for a derivation):

$$p(y_{n+1} \mid \boldsymbol{x}_{n+1}, \boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}) = \mathcal{N}(y_{n+1} \mid \mu_{n+1}, \text{Var}_{n+1}) \tag{2}$$

with mean and variance

$$\mu_{n+1} = \boldsymbol{k}_*^{\mathsf{T}} \left[\boldsymbol{K} + \sigma^2 \cdot \boldsymbol{I_n} \right]^{-1} \boldsymbol{y}_{1:n}$$

$$\text{Var}_{n+1} = k_{**} - \boldsymbol{k}_*^{\mathsf{T}} \left[\boldsymbol{K} + \sigma^2 \boldsymbol{I} \right]^{-1} \boldsymbol{k}_*,$$

where

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ & \ddots & \\ k(\boldsymbol{x}_n, \boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix}$$

$$\boldsymbol{k}_* = \left(k(\boldsymbol{x}_1, \boldsymbol{x}_{n+1}), \ldots, k(\boldsymbol{x}_n, \boldsymbol{x}_{n+1}) \right)^{\mathsf{T}}$$

$$k_{**} = k(\boldsymbol{x}_{n+1}, \boldsymbol{x}_{n+1}) + \sigma^2.$$

The GP equations above assume fixed kernel parameters $\lambda_1, \ldots, \lambda_p$ and fixed observation noise variance σ^2. These constitute the GP's *hyperparameters*. In contrast to hyperparameters in other models, the number of GP hyperparameters grows with the input dimensionality, and their optimization is an integral part of fitting a GP: they are typically set by maximizing the *marginal likelihood* $p(\boldsymbol{y}_{1:n})$ of the data with a gradient-based optimizer (again, see Rasmussen & Williams [94] for details). The choice of optimizer can make a big difference in practice; we used the `minFunc` [101] implementation of a limited-memory version of BFGS [87].

Learning a GP model from data can be computationally expensive. Inverting the $n \times n$ matrix $[\boldsymbol{K} + \sigma^2 \boldsymbol{I_n}]$ takes $O(n^3)$ time and has to be done in every of the h hyperparameter optimization steps, yielding a total complexity of $O(h \cdot n^3)$. Subsequent predictions at a new input require only time $O(n)$ and $O(n^2)$ for the mean and the variance, respectively.

3.5. Regression trees

Regression trees [16] are simple tree-based regression models. They are known to handle discrete inputs well; their first application to the prediction of algorithm performance was by Bartz-Beielstein & Markon [8]. The leaf nodes of regression trees partition the input space into disjoint regions $R_1, \ldots, R_M$, and use a simple model for prediction in each region R_m; the most common choice is to predict a constant c_m. This leads to the following prediction for an input point $\boldsymbol{x}$:

$$\hat{\mu}(\boldsymbol{x}) = \sum_{m=1}^{M} c_m \cdot \mathbb{I}_{\boldsymbol{x} \in R_m},$$

where the indicator function $\mathbb{I}_z$ takes value 1 if the proposition z is true and 0 otherwise. Note that since the regions R_m partition the input space, this sum will always involve exactly one non-zero term. We denote the subset of training data points in region R_m as $\mathcal{D}_m$. Under the standard squared error loss function $\sum_{i=1}^{n} (y_i - \hat{\mu}(\boldsymbol{x}_i))^2$, the error-minimizing choice of constant c_m in region R_m is then the sample mean of the data points in $\mathcal{D}_m$:

$$c_m = \frac{1}{|\mathcal{D}_m|} \sum_{\boldsymbol{x}_i \in R_m} y_i. \tag{3}$$

To construct a regression tree, we use the following standard recursive procedure, which starts at the root of the tree with all available training data points $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$. We consider binary partitionings of a given node's data along *split variables* j and *split points* s. For a real-valued split variable j, s is a scalar and data point $\boldsymbol{x}_i$ is assigned to region $R_1(j, s)$ if $x_{i,j} \leqslant s$ and to region $R_2(j, s)$ otherwise. For a categorical split variable j, s is a set, and data point $\boldsymbol{x}_i$ is assigned to region $R_1(j, s)$ if $x_{i,j} \in s$ and to region $R_2(j, s)$ otherwise. At each node, we select split variable j and split point s to minimize the sum of squared differences to the regions' means,

$$l(j, s) = \sum_{\boldsymbol{x}_i \in R_1(j,s)} (y_i - c_1)^2 + \sum_{\boldsymbol{x}_i \in R_2(j,s)} (y_i - c_2)^2, \tag{4}$$

where c_1 and c_2 are chosen according to Eq. (3) as the sample means in regions $R_1(j, s)$ and $R_2(j, s)$, respectively. We continue this procedure recursively, finding the best split variable and split point, partitioning the data into two child nodes, and recursing into the child nodes. The process terminates when all training data points in a node share the same $\boldsymbol{x}$ values, meaning that no more splits are possible. This procedure tends to overfit data, which can be mitigated by recursively pruning away branches that contribute little to the model's predictive accuracy. We use cost-complexity pruning with 10-fold cross-validation to identify the best tradeoff between complexity and predictive quality; see the book by Hastie et al. [39] for details.

In order to predict the response value at a new input, $\boldsymbol{x}_i$, we *propagate $\boldsymbol{x}$ down the tree*, that is, at each node with split variable j and split point s, we continue to the left child node if $\boldsymbol{x}_{i,j} \leqslant s$ (for real-valued variable j) or $\boldsymbol{x}_{i,j} \in s$ (for categorical variable j), and to the right child node otherwise. The predictive mean for $\boldsymbol{x}_i$ is the constant c_m in the leaf that this process selects; there is no variance predictor.

3.5.1. Complexity of constructing regression trees

If implemented efficiently, the computational cost of fitting a regression tree is small. At a single node with n data points of dimensionality p, it takes $O(p \cdot n \log n)$ time to identify the best combination of split variable and point, because for each continuous split variable j, we can sort the n values $\mathbf{x}_{1,j}, \ldots, \mathbf{x}_{n,j}$ and only consider up to $n-1$ possible split points between different values. The procedure for categorical split variables has the same complexity: we consider each of the variable's k categorical values u_l, compute score $s_l = \text{mean}(\{y_i \mid \boldsymbol{x}_{i,j} = u_l\})$ across the node's data points, sort $(u_1, \ldots, u_k)$ by these scores, and only consider the k binary partitions with consecutive scores in each set. For the squared error loss function we use, the computation of $l(j, s)$ (see Eq. (4)) can be performed in amortized $O(1)$ time for each of j's split points s, such that the total time required for determining the best split point of a single variable is $O(n \log n)$. The complexity of building a regression tree depends on how balanced it is. In the worst case, one data point is split off at a time, leading to a tree of depth $n-1$ and a complexity of $O(p \sum_{i=1}^{n} (n-i) \log (n-i))$, which is $O(p \cdot n^2 \log n)$. In the best case—a balanced tree—we have the recurrence relation $T(n) = v \cdot n \log n + 2T(n/2)$, leading to a complexity of $O(p \cdot n \log^2 n)$. In our experience, trees are not perfectly balanced, but are much closer to the best case than to the worst case. For example, 10 000 data points typically led to tree depths between 25 and 30 (whereas $\log_2(10\,000) \approx 13$).

Prediction with regression trees is cheap; we merely need to propagate new query points $\mathbf{x}_{n+1}$ down the tree. At each node with continuous split variable j and split point s, we only need to compare $\mathbf{x}_{n+1,j}$ to s, an $O(1)$ operation. For categorical split variables, we can store a bit mask of the values in s to enable $O(1)$ member queries. In the worst case (where the tree has depth $n-1$), prediction thus takes $O(n)$ time, and in the best (balanced) case it takes $O(\log n)$ time.

4. New modelling techniques for EPMs

In this section we extend existing modelling techniques for EPMs, with the primary goal of improving runtime predictions for highly parameterized algorithms. The methods described here draw on advanced machine learning techniques, but, to the best of our knowledge, our work is the first to have applied them for algorithm performance prediction. More specifically, we show how to extend all models to handle categorical inputs (required for predictions in partially categorical configuration spaces) and describe two new model families well-suited to modelling the performance of highly parameterized algorithms based on potentially large amounts of data: the projected process approximation to Gaussian processes and random forests of regression trees.

4.1. Handling categorical inputs

Empirical performance models have historically been limited to continuous-valued inputs; the only approach that has so far been used for performance predictions based on discrete-valued inputs is regression trees [8]. In this section, we first present a standard method for encoding categorical parameters as real-valued parameters, and then present a kernel for handling categorical inputs more directly in Gaussian processes.

4.1.1. Extension of existing methods using 1-in-$\mathcal{K}$ encoding

A standard solution for extending arbitrary modelling techniques to handle categorical inputs is the so-called 1-in-$\mathcal{K}$ encoding scheme (see, *e.g.*, [12]), which encodes categorical inputs with finite domain size $\mathcal{K}$ as $\mathcal{K}$ binary inputs. Specifically,

if the ith column of the design matrix $\boldsymbol{X}$ is categorical with domain D_i, we replace it with $|D_i|$ binary indicator columns, where the new column corresponding to each $d \in D_i$ contains values $[\mathbb{I}_{x_{1,i}=d}, \ldots, \mathbb{I}_{x_{n,i}=d}]^\mathsf{T}$; for each data point, exactly one of the new columns is 1, and the rest are all 0. After this transformation, the new columns are treated exactly like the original real-valued columns, and arbitrary modelling techniques for numerical inputs become applicable.

4.1.2. A weighted hamming distance kernel for categorical inputs in GPs

A problem with the 1-in-$\mathcal{K}$ encoding is that using it increases the size of the input space considerably, causing some regression methods to perform poorly. We now define a kernel for handling categorical inputs in GPs more directly. Our kernel is similar to the standard squared exponential kernel of Eq. (1), but instead of measuring the (weighted) squared distance, it computes a (weighted) Hamming distance:

$$K_{\mathrm{cat}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\sum_{l=1}^{p} (-\lambda_l \cdot \mathbb{I}_{x_{i,l} \neq x_{j,l}}) \right). \tag{5}$$

For a combination of continuous and categorical input dimensions $\mathcal{P}_{\mathrm{cont}}$ and $\mathcal{P}_{\mathrm{cat}}$, we combine the two kernels:

$$K_{\mathrm{mixed}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\sum_{l \in \mathcal{P}_{\mathrm{cont}}} \left(-\lambda_l \cdot (x_{i,l} - x_{j,l})^2\right) + \sum_{l \in \mathcal{P}_{\mathrm{cat}}} \left(-\lambda_l \cdot \mathbb{I}_{x_{i,l} \neq x_{j,l}}\right) \right).$$

Although K_{mixed} is a straightforward adaptation of the standard kernel in Eq. (1), we are not aware of any prior use of it. To use this kernel in GP regression, we have to show that it is *positive definite*.

Definition 1 (*Positive definite kernel*). A function $k : \mathcal{I} \times \mathcal{I} \mapsto \mathbb{R}$ is a *positive definite kernel* iff it is (1) *symmetric*: for any pair of inputs $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{I}$, k satisfies $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = k(\boldsymbol{x}_j, \boldsymbol{x}_i)$; and (2) *positive definite*: for any n inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathcal{I}$ and any n constants $c_1, \ldots, c_n \in \mathbb{R}$, k satisfies $\sum_{i=1}^{n} \sum_{j=1}^{n} (c_i \cdot c_j \cdot k(\boldsymbol{x}_i, \boldsymbol{x}_j)) \geqslant 0$.

Proposition 2 (*K_{mixed} is positive definite*). *For any combination of continuous and categorical input dimensions $\mathcal{P}_{\mathrm{cont}}$ and $\mathcal{P}_{\mathrm{cat}}$, K_{mixed} is a positive definite kernel function.*

Appendix B in the online appendix provides the proof, which shows that K_{mixed} can be constructed from simpler positive definite functions, and uses the facts that the space of positive definite kernel functions is closed under addition and multiplication.

Our new kernel can be understood as implicitly performing a 1-in-$\mathcal{K}$ encoding. Note that Kernel K_{mixed} has one hyperparameter λ_i for each input dimension. By using a 1-in-$\mathcal{K}$ encoding and kernel K_{cont} instead, we end up with one hyperparameter λ_i for each *encoded* dimension; if we then reparameterize K_{cont} to share a single hyperparameter λ_l across the encoded dimensions resulting from a single original input dimension l, we recover K_{mixed}.

Since K_{mixed} is rather expressive, one may worry about overfitting. Thus, we also experimented with two variations: (1) sharing the same hyperparameter λ across all input dimensions; and (2) sharing λ_1 across algorithm parameters and λ_2 across instance features. We found that neither variation outperformed K_{mixed}.

4.2. Scaling to large amounts of data with approximate Gaussian processes

The time complexity of fitting Gaussian processes is cubic in the number of data points, which limits the amount of data that can be used in practice to fit these models. To deal with this obstacle, the machine learning literature has proposed various approximations to Gaussian processes (see, *e.g.*, [93]). To the best of our knowledge, these approximate GPs have previously been applied to runtime prediction only in our work on parameter optimization [59] (considering parameterized algorithms, but only single problem instances). We experimented with the Bayesian committee machine [110], the informative vector machine [72], and the projected process (PP) approximation [94]. All of these methods performed similarly, with the PP approximation having a slight edge. Below, we give the equations for the PP's predictive mean and variance; for a derivation, see the Rasmussen & Williams [94].

The PP approximation to GPs uses a subset of a of the n training data points, the so-called *active set*. Let v be a vector consisting of the indices of these a data points. We extend the notation for exact GPs (see Section 3.4) as follows: let K_{aa} denote the a by a matrix with $K_{\mathrm{aa}}(i, j) = k(\boldsymbol{x}_{v(i)}, \boldsymbol{x}_{v(j)})$ and let $\boldsymbol{K}_{\mathrm{an}}$ denote the a by n matrix with $\boldsymbol{K}_{\mathrm{an}}(i, j) = k(\boldsymbol{x}_{v(i)}, \boldsymbol{x}_j)$. The predictive distribution of the PP approximation is then a normal distribution with mean and variance

$$\mu_{n+1} = \boldsymbol{k}_*^\mathsf{T} \left(\sigma^2 \boldsymbol{K}_{\mathrm{aa}} + \boldsymbol{K}_{\mathrm{an}} \boldsymbol{K}_{\mathrm{an}}^\mathsf{T}\right)^{-1} \boldsymbol{K}_{\mathrm{an}} y_{1:n}$$

$$\mathrm{Var}_{n+1} = k_{**} - \boldsymbol{k}_*^\mathsf{T} \boldsymbol{K}_{\mathrm{aa}}^{-1} \boldsymbol{k}_* + \sigma^2 \boldsymbol{k}_*^\mathsf{T} \left(\sigma^2 \boldsymbol{K}_{\mathrm{aa}} + \boldsymbol{K}_{\mathrm{an}} \boldsymbol{K}_{\mathrm{an}}^\mathsf{T}\right)^{-1} \boldsymbol{k}_*.$$

We perform h steps of hyperparameter optimization based on a standard GP, trained using a set of a data points sampled uniformly at random without replacement from the n input data points. We then use the resulting hyperparameters and

another independently sampled set of a data points (sampled in the same way) for the subsequent PP approximation. In both cases, if $a > n$, we only use n data points.

The complexity of the PP approximation is superlinear only in a; therefore, the approach is much faster when we choose $a \ll n$. The hyperparameter optimization based on a data points takes time $O(h \cdot a^3)$. In addition, there is a one-time cost of $O(a^2 \cdot n)$ for evaluating the PP equations. Thus, the time complexity for fitting the approximate GP model is $O([h \cdot a + n] \cdot a^2)$, as compared to $O(h \cdot n^3)$ for the exact GP model. The time complexity for predictions with this PP approximation is $O(a)$ for the mean and $O(a^2)$ for the variance of the predictive distribution [94], as compared to $O(n)$ and $O(n^2)$, respectively, for the exact version. In our experiments, we set $a = 300$ and $h = 50$ to achieve a good compromise between speed and predictive accuracy.

4.3. Random forest models

Regression trees, as discussed in Section 3.5, are a flexible modelling technique that is particularly effective for discrete input data. However, they are also well known to be sensitive to small changes in the data and are thus prone to over-fitting. Random forests [15] overcome this problem by combining multiple regression trees into an ensemble. Known for their strong predictions for high-dimensional and discrete input data, random forests are an obvious choice for runtime predictions of highly parameterized algorithms. Nevertheless, to the best of our knowledge, they have not been used for algorithm runtime prediction except in our own recent work on algorithm configuration [59,55,54,56], which used a proto-type implementation of the models we describe here.[6] In the following, we describe the standard RF framework and some non-standard implementation choices we made.

4.3.1. The standard random forest framework

A random forest (RF) consists of a set of regression trees. If grown to sufficient depths, regression trees are extraordinarily flexible predictors, able to capture very complex interactions and thus having low bias. However, this means they can also have high variance: small changes in the data can lead to a dramatically different tree. Random forests [15] reduce this variance by aggregating predictions across multiple different trees. (This is an alternative to the pruning procedure described previously; thus, the trees in random forests are not pruned, but are rather grown until each node contains no more than $n_{\min}$ data points.) These trees are made to be different by training them on different subsamples of the training data, and/or by permitting only a random subset of the variables as split variables at each node. We chose the latter option, using the full training set for each tree. (We did experiment with a combination of the two approaches, but found that it yielded slightly worse performance.)

Mean predictions for a new input $\boldsymbol{x}$ are trivial: predict the response for $\boldsymbol{x}$ with each tree and average the predictions. The predictive quality improves as the number of trees, B, grows, but computational cost also grows linearly in B. We used $B = 10$ throughout our experiments to keep computational costs low. Random forests have two additional hyperparameters: the percentage of variables to consider at each split point, $perc$, and the minimal number of data points required in a node to make it eligible to be split further, $n_{\min}$. We set $perc = 0.5$ and $n_{\min} = 5$ by default.

4.3.2. Modifications to standard random forests

We introduce a simple, yet effective, method for quantifying predictive uncertainty in random forests. (Our method is similar in spirit to that of Meinshausen [81], who recently introduced quantile regression trees, which allow for predictions of quantiles of the predictive distribution; in contrast, we predict a mean and a variance.) In each leaf of each regression tree, in addition to the empirical mean of the training data associated with that leaf, we store the empirical variance of that data. To avoid making deterministic predictions for leaves with few data points, we round the stored variance up to at least the constant $\sigma^2_{\min}$; we set $\sigma^2_{\min} = 0.01$ throughout. For any input, each regression tree T_b thus yields a predictive mean μ_b and a predictive variance σ^2_b. To combine these estimates into a single estimate, we treat the forest as a mixture model of B different models. We denote the random variable for the prediction of tree T_b as L_b and the overall prediction as L, and then have $L = L_b$ if $Y = b$, where Y is a multinomial variable with $p(Y = i) = 1/B$ for $i = 1, \ldots, B$. The mean and variance for L can then be expressed as:

$$\mu = \mathbb{E}[L] = \frac{1}{B} \sum_{b=1}^{B} \mu_b;$$

$$\sigma^2 = \text{Var}(L) = \mathbb{E}\big[\text{Var}(L|Y)\big] + \text{Var}\big(\mathbb{E}[L|Y]\big)$$

$$= \left(\frac{1}{B} \sum_{b=1}^{B} \sigma^2_b\right) + \big(\mathbb{E}\big[\mathbb{E}(L|Y)^2\big] - \mathbb{E}\big[\mathbb{E}(L|Y)\big]^2\big)$$

---

[6] Note that random forests have also been found to be effective in predicting the approximation ratio of 2-opt on Euclidean TSP instances [82].

$$= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2\right) + \left(\frac{1}{B}\sum_{b=1}^{B}\mu_b^2\right) - \mathbb{E}[L]^2$$

$$= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2 + \mu_b^2\right) - \mu^2.$$

Thus, our predicted mean is simply the mean across the means predicted by the individual trees in the random forest. To compute the variance prediction, we used the law of total variance (see, *e.g.*, [114]), which allows us to write the total variance as the variance across the means predicted by the individual trees (predictions are uncertain if the trees disagree), plus the average variance of each tree (predictions are uncertain if the predictions made by individual trees tend to be uncertain).

A second non-standard ingredient in our models concerns the choice of split points. Consider splits on a real-valued variable j. Note that when the loss in Eq. (4) is minimized by choosing split point s between the values of $\boldsymbol{x}_{k,j}$ and $\boldsymbol{x}_{l,j}$, we are still free to choose the exact location of s anywhere in the interval $(\boldsymbol{x}_{k,j}, \boldsymbol{x}_{l,j})$. Traditionally, s is chosen as the midpoint between $\boldsymbol{x}_{k,j}$ and $\boldsymbol{x}_{l,j}$. Instead, here we draw it uniformly at random from $(\boldsymbol{x}_{k,j}, \boldsymbol{x}_{l,j})$. In the limit of an infinite number of trees, this leads to a linear interpolation of the training data instead of a partition into regions of constant prediction. Furthermore, it causes variance estimates to vary smoothly and to grow with the distance from observed data points.

4.3.3. Complexity of fitting random forests

The computational cost for fitting a random forest is relatively low. We need to fit B regression trees, each of which is somewhat easier to fit than a normal regression tree, since at each node we only consider $v = \max(1, \lfloor perc \cdot p \rfloor)$ out of the p possible split variables. Building B trees simply takes B times as long as building a single tree. Thus—by the same argument as for regression trees—the complexity of learning a random forest is $O(B \cdot v \cdot n^2 \cdot \log n)$ in the worst case (splitting off one data point at a time) and $O(B \cdot v \cdot n \cdot \log^2 n)$ in the best case (perfectly balanced trees). Our random forest implementation is based on a port of Matlab's regression tree code to C, which yielded speedups of between one and two orders of magnitude.

Prediction with a random forest model entails predicting with B regression trees (plus an $O(B)$ computation to compute the mean and variance across those predictions). The time complexity of a single prediction is thus $O(B \cdot n)$ in the worst case and $O(B \cdot \log n)$ for perfectly balanced trees.

5. Problem-specific instance features

While the methods we have discussed so far could be used to model the performance of any algorithm for solving any problem, in our experiments, we investigated specific NP-complete problems. In particular, we considered the propositional satisfiability problem (SAT), mixed integer programming (MIP) problems, and the travelling salesperson problem (TSP). Our reasons for choosing these three problems are as follows. SAT is the prototypical NP-hard decision problem and is thus interesting from a theory perspective; modern SAT solvers are also one of the most prominent approaches in hardware and software verification [92]. MIP is a canonical representation for constrained optimization problems with integer-valued and continuous variables, which serves as a unifying framework for NP-complete problems and combines the expressive power of integrality constraints with the efficiency of continuous optimization. As a consequence, it is very widely used both in academia and industry [61]. Finally, TSP is one of the most widely studied NP-hard optimization problems, and also of considerable interest for industry [21].

We tailor EPMs to a particular problem through the choice of instance features.[7] Here we describe comprehensive sets of features for SAT, MIP, and TSP. For each of these problems, we summarize sets of features found in the literature and introduce many novel features. While all these features are polynomial-time computable, we note that some of them can be computationally expensive for very large instances (*e.g.*, taking cubic time). For some applications such expensive features will be reasonable—in particular, we note that for applications that take features as a one-time input, but build models repeatedly, it can even make sense to use features whose cost exceeds that of solving the instance; examples of such applications include model-based algorithm configuration [55] and complex empirical analyses based on performance predictions [53,57]. In runtime-sensitive applications, on the other hand, it may make sense to use only a subset of the features described here. To facilitate this, we categorize all features into one of four "cost classes": trivial, cheap, moderate, and expensive. In our experimental evaluation, we report the empirical cost of these feature classes and the predictive performance that can be achieved using them (see Table 3 on page 97). We also identify features introduced in this work and quantify their contributions to model performance.

Probing features are a generic family of features that deserves special mention. They are computed by briefly running an existing algorithm for the given problem on the given instance and extracting characteristics from that algorithm's trajectory—an idea closely related to that of landmarking in meta-learning [91]. Probing features can be defined with little

---

[7] If features are unavailable for an NP-complete problem of interest, one alternative is to reduce the problem to SAT, MIP, or TSP—a polynomial-time operation—and then compute some of the features we describe here. We do not expect this approach to be computationally efficient, but do observe that it extends the reach of existing EPM construction techniques to any NP-complete problem.

Problem Size Features:

1–2. **Number of variables and clauses in original formula** (**trivial**): denoted v and c, respectively
3–4. **Number of variables and clauses after simplification with SATElite (cheap)**: denoted v' and c', respectively
5–6. **Reduction of variables and clauses by simplification (cheap)**: $(v-v')/v'$ and $(c-c')/c'$
7. **Ratio of variables to clauses (cheap)**: v'/c'

Variable-Clause Graph Features:

8–12. **Variable node degree statistics (expensive):** mean, variation coefficient, min, max, and entropy
13–17. **Clause node degree statistics (cheap):** mean, variation coefficient, min, max, and entropy

Variable Graph Features (expensive):

18–21. **Node degree statistics:** mean, variation coefficient, min, and max
22–26. **Diameter*:** mean, variation coefficient, min, max, and entropy

Clause Graph Features (expensive):

27–31. **Node degree statistics:** mean, variation coefficient, min, max, and entropy
32–36. **Clustering Coefficient*:** mean, variation coefficient, min, max, and entropy

Balance Features:

37–41. **Ratio of positive to negative literals in each clause (cheap):** mean, variation coefficient, min, max, and entropy
42–46. **Ratio of positive to negative occurrences of each variable (expensive):** mean, variation coefficient, min, max, and entropy
47–49. **Fraction of unary, binary, and ternary clauses (cheap)**

Proximity to Horn Formula (expensive):

50. **Fraction of Horn clauses**
51–55. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max, and entropy

DPLL Probing Features:

56–60. **Number of unit propagations (expensive):** computed at depths 1, 4, 16, 64 and 256

61–62. **Search space size estimate (cheap):** mean depth to contradiction, estimate of the log of number of nodes

LP-Based Features (moderate):

63–66. **Integer slack vector :** mean, variation coefficient, min, and max
67. **Ratio of integer vars in LP solution**
68. **Objective value of LP solution**

Local Search Probing Features, based on 2 seconds of running each of SAPS and GSAT (cheap):

69–78. **Number of steps to the best local minimum in a run:** mean, median, variation coefficient, 10th and 90th percentiles
79–82. **Average improvement to best in a run:** mean and coefficient of variation of improvement per step to best solution
83–86. **Fraction of improvement due to first local minimum:** mean and variation coefficient
87–90. **Best solution:** mean and variation coefficient

Clause Learning Features* (based on 2 seconds of running ZCHAFF_RAND; cheap):

91–99. **Number of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles
100–108. **Length of learned clause:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Survey Propagation Features* (moderate)

109–117. **Confidence of survey propagation:** For each variable, compute the higher of $P(true)/P(false)$ or $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles
118–126. **Unconstrained variables:** For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Timing Features*

127–138. **CPU time required for feature computation:** one feature for each of 12 subsets of features (see text for details)

Fig. 1. SAT instance features. New features are marked with *.

effort for a wide variety of problems; indeed, in earlier work, we introduced the first probing features for SAT [90] and showed that probing features based on one type of algorithm (*e.g.*, local search) are often useful for predicting the performance of another type of algorithm (*e.g.*, tree search). Here we introduce the first probing features for MIP and TSP. Another new, generic family of features are *timing features*, which measure the time other groups of features take to compute. Code and binaries for computing all our features, along with documentation providing additional details, are available online at http://www.cs.ubc.ca/labs/beta/Projects/EPMs/.

5.1. Features for Propositional Satisfiability (SAT)

Fig. 1 summarizes 138 features for SAT. Since various preprocessing techniques are routinely used before applying a general-purpose SAT solver and typically lead to substantial reductions in instance size and difficulty (especially for industrial-like instances), we apply the preprocessing procedure SATElite [23] on all instances first, and then compute instance features on the preprocessed instances. The first 90 features, with the exception of features 22–26 and 32–36, were introduced in our previously published work on SATzilla [90,119]. They can be categorized as *problem size* features (1–7), *graph-based* features (8–36), *balance* features (37–49), *proximity to Horn formula* features (50–55), *DPLL probing* features (56–62), *LP-based* features (63–68), and *local search probing* features (69–90).

Our new features (devised over the last five years in our ongoing work on SATzilla and so far only mentioned in short solver descriptions [118,121]) fall into four categories. First, we added two additional subgroups of graph-based features. Our new *diameter* features 22–26 are based on the variable graph [41]. For each node *i* in that graph, we compute the longest shortest path between *i* and any other node. As with most of the features that follow, we then compute various statistics

over this vector (*e.g.*, mean, max); we do not state the exact statistics for each vector below but list them in Fig. 1. Our new *clustering coefficient* features 32–36 measure the local cliqueness of the clause graph. For each node in the clause graph, let p denote the number of edges present between the node and its neighbours, and let m denote the maximum possible number of such edges; we compute p/m for each node.

Second, our new *clause learning* features (91–108) are based on statistics gathered in 2-second runs of Zchaff_rand [80]. We measure the number of learned clauses (features 91–99) and the length of the learned clauses (features 100–108) after every 1000 search steps. Third, our new *survey propagation* features (109–126) are based on estimates of variable bias in a SAT formula obtained using probabilistic inference [46]. We used VARSAT's implementation to estimate the probabilities that each variable is true in every satisfying assignment, false in every satisfying assignment, or unconstrained. Features 109–117 measure the confidence of survey propagation (that is, $\max(P_{\text{true}}(i)/P_{\text{false}}(i), P_{\text{false}}(i)/P_{\text{true}}(i))$ for each variable i) and features 118–126 are based on the $P_{\text{unconstrained}}$ vector.

Finally, our new *timing features* (127–138) measure the time taken by 12 different blocks of feature computation code: instance preprocessing by SATElite, problem size (1–6), variable-clause graph (clause node) and balance features (7, 13–17, 37–41, 47–49); variable-clause graph (variable node), variable graph and proximity to Horn formula features (8–12, 18–21, 42–46, 50–55); diameter-based features (22–26); clause graph features (27–36); unit propagation features (56–60); search space size estimation (61–62); LP-based features (63–68); local search probing features (69–90) with SAPS and GSAT; clause learning features (91–108); and survey propagation features (109–126).

5.2. Features for mixed integer programs

Fig. 2 summarizes 121 features for mixed integer programs (*i.e.*, MIP instances). These include 101 features based on existing work [76,48,67], 15 new *probing* features, and 5 new *timing* features. Features 1–101 are primarily based on features for the combinatorial winner determination problem from our past work [76], generalized to MIP and previously only described in a Ph.D. thesis [48]. These features can be categorized as *problem type & size* features (1–25), *variable-constraint graph* features (26–49), *linear constraint matrix* features (50–73), *objective function* features (74–91), and *LP-based* features (92–95). We also integrated ideas from the feature set used by Kadioglu et al. [67] (*right-hand side* features (96–101) and the computation of separate statistics for continuous variables, non-continuous variables, and their union). We extended existing features by adding richer statistics where applicable: medians, variation coefficients (vc), and percentile ratios (q90/q10) of vector-based features.

We introduce two new sets of features. Firstly, our new *MIP probing features* 102–116 are based on 5-second runs of CPLEX with default settings. They are obtained via the CPLEX API and include 6 *presolving* features based on the output of CPLEX's presolving phase (102–107); 5 *probing cut usage* features describing the different cuts CPLEX used during probing (108–112); and 4 *probing result* features summarizing probing runs (113–116). Secondly, our new *timing features* 117–121 capture the CPU time required for computing five different groups of features: variable-constraint graph, linear constraint matrix, and objective features for three subsets of variables ("continuous", "non-continuous", and "all", 26–91); LP-based features (92–95); and CPLEX probing features (102–116). The cost of computing the remaining features (1–25, 96–101) is small (linear in the number of variables or constraints).

5.3. Features for the travelling salesperson problem (TSP)

Fig. 3 summarizes 64 features for the travelling salesperson problem (TSP). Features 1–50 are new, while features 51–64 were introduced by Smith-Miles et al. [108]. Features 51–64 capture the spatial distribution of nodes (features 51–61) and clustering of nodes (features 62–64); we used the authors' code (available at http://www.vanhemert.co.uk/files/TSP-feature-extract-20120212.tar.gz) to compute these features.

Our 50 new TSP features are as follows.[8] The *problem size* feature (1) is the number of nodes in the given TSP. The *cost matrix* features (2–4) are statistics of the cost between two nodes. Our *minimum spanning tree* features (5–11) are based on constructing a minimum spanning tree over all nodes in the TSP: features 5–8 are the statistics of the edge costs in the tree and features 9–11 are based on its node degrees. Our *cluster distance* features (12–14) are based on the cluster distance between every pair of nodes, which is the minimum bottleneck cost of any path between them; here, the bottleneck cost of a path is defined as the largest cost along the path. Our *local search probing* features (15–32) are based on 20 short runs (1000 steps each) of LK [78], using the implementation available from [22]. Specifically, features 15–17 are based on the tour length obtained by LK; features 18–20, 21–23, and 24–26 are based on the tour length of local minima, the tour quality improvement per search step, and the number of search steps to reach a local minimum, respectively; features 27–29 measure the Hamming distance between two local minima; and features 30–32 describe the probability of edges appearing in any local minimum encountered during probing. Our *branch and cut probing* features (33–43) are based on 2-second runs of Concorde. Specifically, features 33–35 measure the improvement of lower bound per cut; feature 36 is the ratio of upper and lower bound at the end of the probing run; and features 37–43 analyze the final LP solution. Feature 44 is the autocorrelation coefficient: a measure of the ruggedness of the search landscape, based on an uninformed random

---

[8] In independent work, Mersmann et al. [82] have introduced feature sets similar to some of those described here.

Problem Type (trivial):

1. **Problem type**: LP, MILP, FIXEDMILP, QP, MIQP, FIXEDMIQP, MIQP, QCP, or MIQCP, as attributed by CPLEX

Problem Size Features (trivial):

2–3. **Number of variables and constraints**: denoted n and m, respectively
  4. **Number of non-zero entries in the linear constraint matrix, A**
5–6. **Quadratic variables and constraints**: number of variables with quadratic constraints and number of quadratic constraints
  7. **Number of non-zero entries in the quadratic constraint matrix, Q**
8–12. **Number of variables of type**: Boolean, integer, continuous, semi-continuous, semi-integer
13–17. **Fraction of variables of type** (summing to 1): Boolean, integer, continuous, semi-continuous, semi-integer
18–19. **Number and fraction of non-continuous variables** (counting Boolean, integer, semi-continuous, and semi-integer variables)
20–21. **Number and fraction of unbounded non-continuous variables**: fraction of non-continuous variables that has infinite lower or upper bound
22–25. **Support size**: mean, median, vc, q90/10 for vector composed of the following values for bounded variables: domain size for binary/integer, 2 for semi-continuous, 1+domain size for semi-integer variables.

Variable-Constraint Graph Features (cheap): each feature is replicated three times, for $X \in \{C, NC, V\}$

26–37. **Variable node degree statistics**: characteristics of vector $(\sum_{c_j \in C} \mathbb{I}(A_{i,j} \neq 0))_{x_i \in X}$: mean, median, vc, q90/10
38–49. **Constraint node degree statistics**: characteristics of vector $(\sum_{x_i \in X} \mathbb{I}(A_{i,j} \neq 0))_{c_j \in C}$: mean, median, vc, q90/10

Linear Constraint Matrix Features (cheap): each feature is replicated three times, for $X \in \{C, NC, V\}$

50–55. **Variable coefficient statistics**: characteristics of vector $(\sum_{c_j \in C} A_{i,j})_{x_i \in X}$: mean, vc
56–61. **Constraint coefficient statistics**: characteristics of vector $(\sum_{x_i \in X} A_{i,j})_{c_j \in C}$: mean, vc
62–67. **Distribution of normalized constraint matrix entries, $A_{i,j}/b_i$**: mean and vc (only of elements where $b_i \neq 0$)

68–73. **Variation coefficient of normalized absolute non-zero entries per row** (the normalization is by dividing by sum of the row's absolute values): mean, vc

Objective Function Features (cheap): each feature is replicated three times, for $X \in \{C, NC, V\}$

74–79. **Absolute objective function coefficients** $\{|c_i|\}_{i=1}^{n}$: mean and stddev
80–85. **Normalized absolute objective function coefficients** $\{|c_i|/n_i\}_{i=1}^{n}$, where n_i denotes the number of non-zero entries in column i of A: mean and stddev
86–91. **squareroot-normalized absolute objective function coefficients** $\{|c_i|/\sqrt{n_i}\}_{i=1}^{n}$: mean and stddev

LP-Based Features (expensive):

92–94. **Integer slack vector:** mean, max, L_2 norm
  95. **Objective function value of LP solution**

Right-hand Side Features (trivial):

96–97. **Right-hand side for $\leqslant$ constraints**: mean and stddev
98–99. **Right-hand side for $=$ constraints**: mean and stddev
100–101. **Right-hand side for $\geqslant$ constraints**: mean and stddev

Presolving Features* (moderate):

102-103. **CPU times**: presolving and relaxation CPU time
104-107. **Presolving result features:** # of constraints, variables, non-zero entries in the constraint matrix, and clique table inequalities after presolving.

Probing Cut Usage Features* (moderate):

108-112. **Number of specific cuts**: clique cuts, Gomory fractional cuts, mixed integer rounding cuts, implied bound cuts, flow cuts

Probing Result features* (moderate):

113-116. **Performance progress**: MIP gap achieved, # new incumbent found by primal heuristics, # of feasible solutions found, # of solutions or incumbents found

Timing Features*

117–121. **CPU time required for feature computation:** one feature for each of 5 groups of features (see text for details)

Fig. 2. MIP instance features; for the variable-constraint graph, linear constraint matrix, and objective function features, each feature is computed with respect to three subsets of variables: continuous, C, non-continuous, NC, and all, V. Features introduced for the first time are marked with *.

walk (see, *e.g.*, [42]). Finally, our *timing features* 45–50 measure the CPU time required for computing feature groups 2–7 (the cost of computing the number of nodes can be ignored).

6. Performance predictions for new instances

We now study the performance of the models described in Sections 3 and 4 (see Table 1 for an overview), based on (various subsets of) the features described in Section 5. In this section, we consider the (only) problem considered by most past work: predicting the performance achieved by the default configuration of a given algorithm on new instances. (We go on to consider making predictions for novel algorithm configurations in Sections 7 and 8.) For brevity, we only present representative empirical results. The full results of our experiments are available in an online appendix at http://www.cs.ubc.ca/labs/beta/Projects/EPMs. All of our data, features, and source code for replicating our experiments is available from the same site.

6.1. Instances and solvers

For SAT, we used a wide range of instance distributions: INDU, HAND, and RAND are collections of industrial, handmade, and random instances from the international SAT competitions and races, and COMPETITION is their union; SWV and IBM are sets of software and hardware verification instances, and SWV-IBM is their union; RANDSAT is a subset of RAND

Problem Size Features* **(trivial):**

   1. **Number of nodes**: denoted n

Cost Matrix Features* **(trivial):**

2–4. **Cost statistics:** mean, variation coefficient, skew

Minimum Spanning Tree Features* **(trivial):**

 5–8. **Cost statistics:** sum, mean, variation coefficient, skew
9–11. **Node degree statistics:** mean, variation coefficient, skew

Cluster Distance Features* **(moderate):**

12–14. **Cluster distance:** mean, variation coefficient, skew

Local Search Probing Features* **(expensive):**

15–17. **Tour cost from construction heuristic:** mean, variation coefficient, skew
18–20. **Local minimum tour length:** mean, variation coefficient, skew
21–23. **Improvement per step:** mean, variation coefficient, skew
24–26. **Steps to local minimum**: mean, variation coefficient, skew
27–29. **Distance between local minima**: mean, variation coefficient, skew
30–32. **Probability of edges in local minima**: mean, variation coefficient, skew

Branch and Cut Probing Features* **(moderate):**

33–35. **Improvement per cut:** mean, variation coefficient, skew
36. **Ratio of upper bound and lower bound**
37–43. **Solution after probing:** Percentage of integer values and non-integer values in the final solution after probing. For non-integer values, we compute statics across nodes: min,max, 25%,50%, 75% quantiles

Ruggedness of Search Landscape* **(cheap):**

44. **Autocorrelation coefficient**

Timing Features*

45–50. **CPU time required for feature computation:** one feature for each of 6 groups (see text)

Node Distribution Features (after instance normalization, moderate)

51. **Cost matrix standard deviation:** standard deviation of cost matrix after instance has been normalized to the rectangle $[(0, 0), (400, 400)]$.
52–55. **Fraction of distinct distances**: precision to 1, 2, 3, 4 decimal places
56–57. **Centroid:** the (x, y) coordinates of the instance centroid
58. **Radius:** the mean distances from each node to the centroid
59. **Area:** the are of the rectangle in which nodes lie
60–61. **nNNd:** the standard deviation and coefficient variation of the normalized nearest neighbour distance
62–64. **Cluster:** #clusters / n, #outliers / n, variation of #nodes in clusters

Fig. 3. TSP instance features. Features introduced for the first time are marked with *.

Table 1
Overview of our models.

Abbreviation	Reference Section	Description
RR	3.2	Ridge regression with 2-phase forward selection
SP	3.2	SPORE-FoBa (ridge regression with forward–backward selection)
NN	3.3	Feed-forward neural network with one hidden layer
PP	4.2	Projected process (approximate Gaussian process)
RT	3.5	Regression tree with cost-complexity pruning
RF	4.3	Random forest

containing only satisfiable instances. We give more details about these distributions in Appendix A.1. For all distributions except RANDSAT, we ran the popular tree search solver, Minisat 2.0 [24]. For INDU, SWV and IBM, we also ran two additional solvers: CryptoMinisat [109] (which won SAT Race 2010 and received gold and silver medals in the 2011 SAT Competition) and SPEAR [5] (which has shown state-of-the-art performance on IBM and SWV with optimized parameter settings [49]). Finally, to evaluate predictions for local search algorithms, we used the RANDSAT instances, and considered two solvers: tnm [112] (which won the random satisfiable category of the 2009 SAT Competition) and the dynamic local search algorithm SAPS [60] (a baseline).

For MIP, we used two instance distributions from computational sustainability (RCW and CORLAT), one from winner determination in combinatorial auctions (REG), two unions of these (CR := CORLAT ∪ RCW and CRR := CORLAT ∪ REG ∪ RCW), and a large and diverse set of publicly available MIP instances (BIGMIX). Details about these distributions are given in Appendix A.2. We used the two state-of-the-art commercial solvers CPLEX [62] and Gurobi [35] (versions 12.1 and 2.0, respectively) and the two strongest non-commercial solvers, SCIP [11] and lp_solve [10] (versions 1.2.1.4 and 5.5, respectively).

For TSP, we used three instance distributions (detailed in Appendix A.3): random uniform Euclidean instances (RUE), random clustered Euclidean instances (RCE), and TSPLIB, a heterogeneous set of prominent TSP instances. On these instance sets, we ran the state-of-the-art systematic and local search algorithms, Concorde [2] and LK-H [40]. For the latter, we computed runtimes as the time required to find an optimal solution.

6.2. Experimental setup

To collect algorithm runtime data, for each algorithm–distribution pair, we executed the algorithm using default parameters on all instances of the distribution, measured its runtimes, and collected the results in a database. All algorithm runs were executed on a cluster of 55 dual 3.2 GHz Intel Xeon PCs with 2 MB cache and 2 GB RAM, running OpenSuSE

Linux 11.1; runtimes were measured as CPU time on these reference machines. We terminated each algorithm run after one CPU hour; this gave rise to *capped* runtime observations, because for each run that was terminated in this fashion, we only observed a lower bound on the runtime. Like most past work on runtime modelling, we simply counted such capped runs as having taken one hour. (In Section 9 we investigate alternatives and conclude that a better treatment of capped runtime data improves predictive performance for our best-performing model.) Basic statistics of the resulting runtime distributions are given in Table 3; Table C.1 in the online appendix lists all the details.

We evaluated different model families by building models on a subset of the data and assessing their performance on data that had not been used to train the models. This can be done visually (as, *e.g.*, in the scatterplots in Fig. 4 on page 96, which show cross-validated predictions for a random subset of up to 1000 data points), or quantitatively. We considered three complementary quantitative metrics to evaluate mean predictions $\mu_1, \ldots, \mu_n$ and predictive variances $\sigma_1^2, \ldots, \sigma_n^2$ given true performance values $y_1, \ldots, y_n$. *Root mean squared error* (*RMSE*) is defined as $\sqrt{1/n \sum_{i=1}^{n} (y_i - \mu_i)^2}$; *Pearson's correlation coefficient* (*CC*) is defined as $(\sum_{i=1}^{n} (\mu_i y_i) - n \cdot \bar{\mu} \cdot \bar{y})/((n-1) \cdot s_\mu \cdot s_y)$, where $\bar{x}$ and s_x denote sample mean and standard deviation of x; and *log likelihood* (*LL*) is defined as $\sum_{i=1}^{n} \log \varphi(\frac{y_i - \mu_i}{\sigma_i})$, where φ denotes the probability density function (PDF) of a standard normal distribution. Intuitively, LL is the log probability of observing the true values y_i under the predicted distributions $\mathcal{N}(\mu_i, \sigma_i^2)$. For CC and LL, higher values are better, while for RMSE lower values are better. We used 10-fold cross-validation and report means of these measures across the 10 folds. We assessed the statistical significance of our findings using a Wilcoxon signed-rank test (we use this paired test, since cross-validation folds are correlated).

6.3. Predictive quality

Table 2 provides quantitative results for all benchmarks, and Fig. 4 visualizes results. At the broadest level, we can conclude that most of the methods were able to capture enough about algorithm performance on training data to make meaningful predictions on test data, most of the time: easy instances tended to be predicted as being easy, and hard ones as being hard. Take, for example the case of predicting the runtime of `Minisat 2.0` on a heterogeneous mix of SAT competition instances (see the leftmost column in Fig. 2 and the top row of Table 2). `Minisat 2.0` runtimes varied by almost six orders of magnitude, while predictions with the better models rarely were off by more than one order of magnitude (outliers may draw the eye in the scatterplot, but quantitatively, the RMSE for predicting $\log_{10}$ runtime was low—*e.g.*, 0.47 for random forests, which means an average misprediction of a factor of $10^{0.47} < 3$). While the models were certainly not perfect, note that even the relatively poor predictions of ridge regression variant RR tended to be accurate within about an order of magnitude, which was enough to enable the portfolio-based algorithm selector `SATzilla` [119] to win five medals in each of the 2007 and 2009 SAT competitions. (Switching to random forest models after 2009 further improved `SATzilla`'s performance [120].)

In our experiments, random forests were the overall winner among the different methods, yielding the best predictions in terms of all our quantitative measures.[9] For SAT, random forests were always the best method, and for MIP they yielded the best performance for the most heterogeneous instance set, `BIGMIX` (see Column 2 of Fig. 4). We attribute the strong performance of random forests on highly heterogeneous data sets to the fact that, as a tree-based approach, they can model very different parts of the data separately; in contrast, the other methods allow the fit in a given part of the space to be influenced more by data in distant parts of the space. Indeed, the ridge regression variants made extremely bad predictions for some outlying points on `BIGMIX`. For the more homogeneous MIP data sets, either random forests or projected processes performed best, often followed closely by ridge regression variant RR. The performance of `CPLEX` on set `RCW` was a special case in that it could be predicted extremely well by all models (see Column 3 of Fig. 4). Finally, for TSP, projected processes and ridge regression had a slight edge for the homogeneous `RUE` and `RCE` benchmarks, whereas tree-based methods (once again) performed best on the most heterogeneous benchmark, `TSPLIB`. The last column of Fig. 4 shows that, in the case where random forests performed worst, the qualitative differences in predictions were small. In terms of computational requirements, random forests were among the cheapest methods, taking between 0.1 and 11 seconds to learn a model.

6.4. Results based on different classes of instance feature

While the previous experiments focused on the performance of the various models based on our entire feature set, we now study the performance of different subsets of features when using the overall best-performing model, random forests. Table 3 presents the results and also lists the cost of the various feature subsets (which in most cases is much smaller than the runtime of the algorithm being modelled). On the broadest level, we note that predictive performance improved as we used more computationally expensive features: *e.g.*, while the trivial features were basically free, they yielded rather poor performance, whereas using the entire feature set almost always led to the best performance. Interestingly, however, for all SAT benchmarks, using at most moderately expensive features yielded results statistically insignificantly different from the

---

[9] For brevity, we only report RMSE values in the tables here; comparative results for correlation coefficients and log likelihoods, given in Table D.3 in the online appendix, are qualitatively similar.

Table 2

Quantitative comparison of models for runtime predictions on previously unseen instances. We report 10-fold cross-validation performance. Lower RMSE values are better (0 is optimal). Note the very large RMSE values for ridge regression on some data sets (we use scientific notation, denoting "$\times 10^x$" as "Ex"); these large errors are due to extremely small/large predictions for a few data points. Boldface indicates performance not statistically significantly different from the best method in each row.

Domain	RMSE						Time to learn model (s)					
	RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
`Minisat 2.0-COMPETITION`	1.01	1.25	0.62	0.92	0.68	**0.47**	**6.8**	28.08	21.84	46.56	20.96	22.42
`Minisat 2.0-HAND`	1.05	1.34	0.63	0.85	0.75	**0.51**	**3.7**	7.92	6.2	44.14	6.15	5.98
`Minisat 2.0-RAND`	0.64	0.76	**0.38**	0.55	0.5	**0.37**	**4.46**	7.98	10.81	46.09	7.15	8.36
`Minisat 2.0-INDU`	0.94	1.01	0.78	0.86	0.71	**0.52**	**3.68**	7.82	5.57	48.12	6.36	4.42
`Minisat 2.0-SWV-IBM`	0.53	0.76	0.32	0.52	0.25	**0.17**	3.51	6.35	4.68	51.67	4.8	**2.78**
`Minisat 2.0-IBM`	0.51	0.71	0.29	0.34	0.3	**0.19**	3.2	5.17	2.6	46.16	2.47	**1.5**
`Minisat 2.0-SWV`	0.35	0.31	0.16	0.1	0.1	**0.08**	3.06	4.9	2.05	53.11	2.37	**1.07**
`CryptoMinisat-INDU`	0.94	0.99	0.94	0.9	0.91	**0.72**	**3.65**	7.9	5.37	45.82	5.03	4.14
`CryptoMinisat-SWV-IBM`	0.77	0.85	0.66	0.83	0.62	**0.48**	3.5	10.83	4.49	48.99	4.75	**2.78**
`CryptoMinisat-IBM`	0.65	0.96	0.55	0.56	0.53	**0.41**	3.19	4.86	2.59	44.9	2.41	**1.49**
`CryptoMinisat-SWV`	0.76	0.78	0.71	0.66	0.63	**0.51**	3.09	4.62	2.09	53.85	2.32	**1.03**
`SPEAR-INDU`	0.95	0.97	0.85	0.87	0.8	**0.58**	**3.55**	9.53	5.4	45.47	5.52	4.25
`SPEAR-SWV-IBM`	0.67	0.85	0.53	0.78	0.49	**0.38**	3.49	6.98	4.32	48.48	4.9	**2.82**
`SPEAR-IBM`	0.6	0.86	0.48	0.66	0.5	**0.38**	3.18	5.77	2.58	45.72	2.5	**1.56**
`SPEAR-SWV`	0.49	0.58	0.48	0.44	**0.47**	0.34	3.09	6.24	2.09	56.09	2.38	**1.13**
`tnm-RANDSAT`	1.01	1.05	**0.94**	**0.93**	1.22	**0.88**	**3.79**	8.63	6.57	46.21	7.64	5.42
`SAPS-RANDSAT`	0.94	1.09	0.73	0.78	0.86	**0.66**	**3.81**	8.54	6.62	49.33	6.59	5.04
`CPLEX-BIGMIX`	2.7E8	0.93	1.02	1	0.85	**0.64**	**3.39**	8.27	4.75	41.25	5.33	**3.54**
`Gurobi-BIGMIX`	1.51	**1.23**	1.41	1.26	1.43	**1.17**	**3.35**	5.12	4.55	40.72	5.45	3.69
`SCIP-BIGMIX`	4.5E6	0.88	0.86	0.91	0.72	**0.57**	**3.43**	**5.35**	4.48	39.51	5.08	**3.75**
`lp_solve-BIGMIX`	1.1	0.9	0.68	1.07	0.63	**0.5**	3.35	4.68	4.62	43.27	**2.76**	4.92
`CPLEX-CORLAT`	0.49	0.52	0.53	**0.46**	0.62	**0.47**	3.19	7.64	5.5	27.54	4.77	**3.4**
`Gurobi-CORLAT`	**0.38**	0.44	0.41	**0.37**	0.51	**0.38**	**3.21**	5.23	5.52	28.58	4.71	**3.31**
`SCIP-CORLAT`	0.39	0.41	0.42	**0.37**	0.5	**0.38**	3.2	7.96	5.52	26.89	5.12	3.52
`lp_solve-CORLAT`	**0.44**	0.48	**0.44**	**0.45**	0.54	**0.41**	3.25	5.06	5.49	31.5	**2.63**	4.42
`CPLEX-RCW`	0.25	0.29	0.1	0.03	0.05	**0.02**	3.11	7.53	5.25	25.84	4.81	**2.66**
`CPLEX-REG`	**0.38**	**0.39**	0.44	**0.38**	0.54	0.42	**3.1**	6.48	5.28	24.95	4.56	3.65
`CPLEX-CR`	0.46	0.58	0.46	**0.43**	0.58	0.45	**4.25**	11.86	11.19	29.92	11.44	8.35
`CPLEX-CRR`	0.44	0.54	0.42	**0.37**	0.47	0.36	**5.4**	18.43	17.34	35.3	20.36	13.19
`LK-H-RUE`	**0.61**	0.63	0.64	**0.61**	0.89	0.67	4.14	**1.14**	12.78	22.95	11.49	11.14
`LK-H-RCE`	**0.71**	0.72	0.75	**0.71**	1.02	0.76	4.19	**2.7**	12.93	24.78	11.54	10.79
`LK-H-TSPLIB`	9.55	**1.11**	1.77	**1.3**	**1.21**	**1.06**	1.61	3.02	0.51	4.3	0.17	**0.11**
`Concorde-RUE`	**0.41**	0.43	0.43	**0.42**	0.59	0.45	4.18	**3.6**	12.7	22.28	10.79	9.9
`Concorde-RCE`	**0.33**	0.34	0.34	0.34	0.46	0.35	4.17	**2.32**	12.68	24.8	11.16	10.18
`Concorde-TSPLIB`	120.6	**0.69**	0.99	**0.87**	0.64	**0.52**	1.54	2.66	0.47	4.26	0.22	**0.12**

best, with substantial reductions in feature computation time. The same was even true for several SAT benchmarks when considering at most cheap features. Our new features clearly showed value: for example, our cheap feature set yielded similar predictive performance as the set of previous features at a much lower cost; and our moderate feature set tended to yield better performance than the previous one at comparable cost. Our new features led to especially clear improvements for MIP, yielding significantly better predictive performance than the previous features in 11/12 cases. Similarly, for TSP, our new features improved performance significantly in 4/6 cases (TSPLIB was too small to achieve reliable results in the two remaining cases, with even the trivial features performing insignificantly worse than the best).

6.5. Impact of hyperparameter optimization

Table 4 shows representative results for the optimization of hyperparameters: it improved robustness somewhat for the ridge regression methods (decreasing the number of extreme outlier predictions) and improved most models slightly across the board. However, these improvements came at the expense of dramatically slower training.[10] In practice, the small improvements in predictive performance that can be obtained via hyperparameter optimization appear likely not to justify this drastic increase in computational cost (*e.g.*, consider model-based algorithm configuration procedures, which iterate

---

[10] Although we fixed the number of hyperparameter optimization steps, variation in model parameters affected learning time more for some model families than for others; for SP, slowdowns reached up to a factor of 3000 (dataset `Minisat 2.0-RAND`).
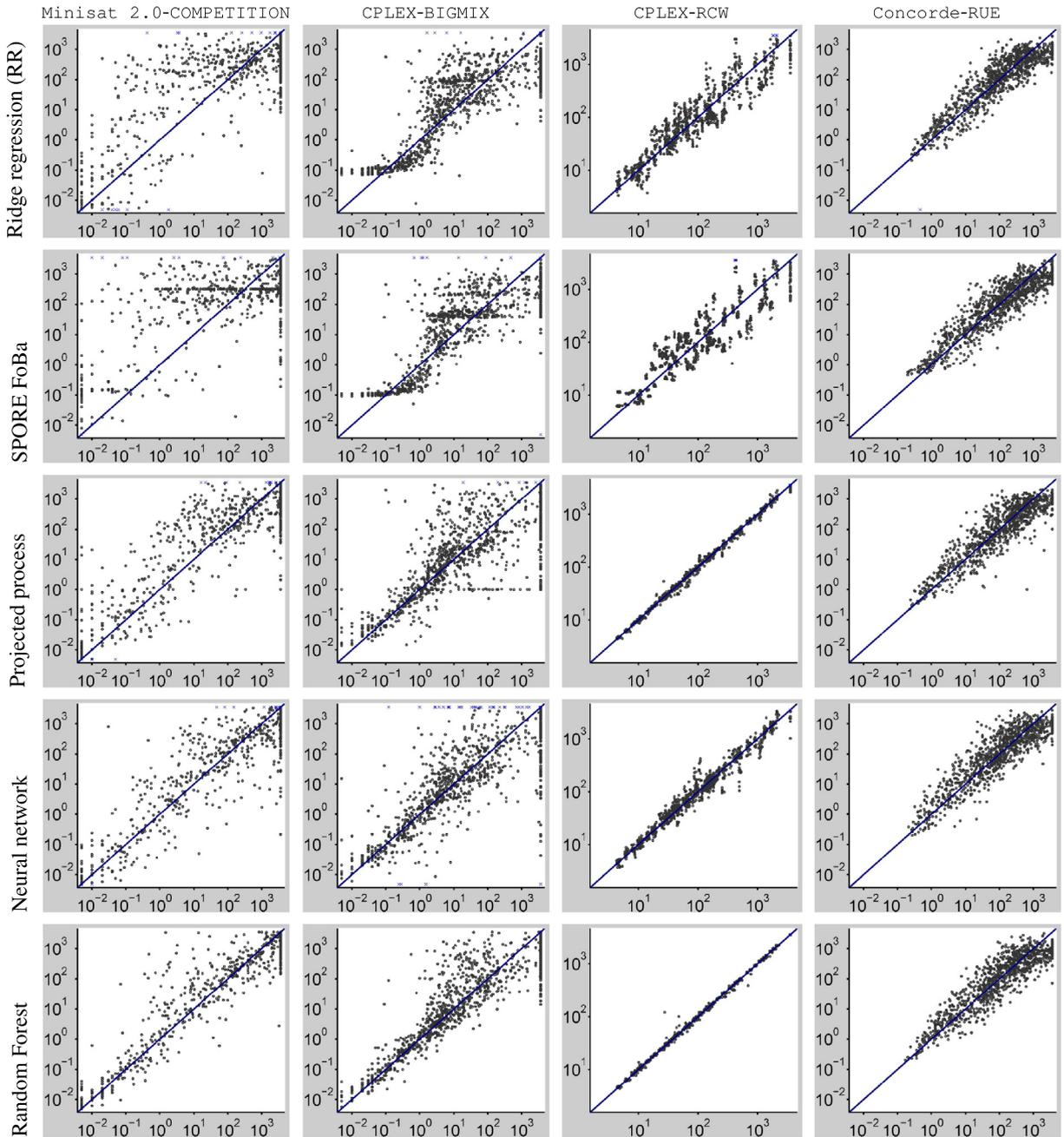
Fig. 4. Visual comparison of models for runtime predictions on previously unseen test instances. The data sets used in each column are shown at the top. The *x*-axis of each scatter plot denotes true runtime and the *y*-axis 2-fold cross-validated runtime as predicted by the respective model; each dot represents one instance. Predictions above 3000 or below 0.001 are denoted by a blue cross rather than a black dot. Figs. D.1–D.10 (in the online appendix) show equivalent plots for the other benchmarks and also include regression trees (whose predictions were similar to those of random forests but had larger spread).

between model construction and data gathering, constructing thousands of models during typical algorithm configuration runs [55]). Thus, we evaluate model performance based on fixed default hyperparameters in the rest of this article. For completeness, our online appendix reports analogous results for models with optimized hyperparameters.

6.6. Predictive quality with sparse training data

We now study how the performance of EPM techniques changes based on the quantity of training data available. Fig. 5 visualizes this relationship for six representative benchmarks; data for all benchmarks appears in the online appendix. Here

Table 3

Quantitative comparison of random forests based on different feature subsets: 'prev' = previous features only; 'mod' = moderate; 'exp' = expensive. Feature sets 'cheap', 'mod', and 'exp' include all cheaper features; *e.g.*, 'exp' uses the entire feature set. We report 10-fold cross-validation performance. Lower RMSE values are better (0 is optimal). Boldface denotes results not statistically significantly different from the best. (Note that depending on the variance, results can have the same rounded mean but still be statistically significantly different; this happens in the cases of CPLEX-RCW and LK-H-RUE.)

Scenario	Alg. runtime		RMSE					Avg. feature time [s]				Max. feature time [s]			
	avg	max	trivial	prev	cheap	mod	exp	prev	cheap	mod	exp	prev	cheap	mod	exp
Minisat 2.0-COMPETITION	2009	3600	1.01	**0.5**	**0.49**	**0.47**	**0.47**	102	**21**	59	109	6E3	**1E3**	1E3	6E3
Minisat 2.0-HAND	1903	3600	1.25	**0.57**	**0.53**	**0.52**	**0.51**	74	**14**	48	79	3E3	**96**	179	3E3
Minisat 2.0-RAND	2497	3600	0.82	**0.39**	**0.38**	**0.37**	**0.37**	59	**23**	52	65	221	**35**	145	230
Minisat 2.0-INDU	1146	3600	0.94	0.58	0.57	**0.55**	0.52	222	**24**	85	231	6E3	**1E3**	1E3	6E3
Minisat 2.0-SWV-IBM	466	3600	0.85	**0.16**	0.17	0.17	0.17	98	**8.4**	59	102	1E3	**74**	153	1E3
Minisat 2.0-IBM	834	3600	1.1	**0.21**	0.25	**0.21**	**0.19**	130	**11**	78	136	1E3	**74**	153	1E3
Minisat 2.0-SWV	0.89	5.32	0.25	**0.08**	0.09	**0.08**	**0.08**	57	**4.9**	34	59	217	**17**	123	226
CryptoMinisat-INDU	1921	3600	1.1	0.81	**0.73**	**0.74**	**0.72**	222	**24**	85	231	6E3	**1E3**	1E3	6E3
CryptoMinisat-SWV-IBM	873	3600	1.07	**0.47**	0.5	**0.49**	**0.48**	98	**8.4**	59	102	1081	**74**	153	1103
CryptoMinisat-IBM	1178	3600	1.2	**0.42**	0.45	**0.42**	**0.41**	130	**11**	78	136	1081	**74**	153	1103
CryptoMinisat-SWV	486	3600	0.89	**0.51**	0.53	**0.49**	**0.51**	57	**4.9**	34	59	217	**17**	123	226
SPEAR-INDU	1685	3600	1.01	0.67	0.62	**0.61**	0.58	222	**24**	85	231	6E3	**1E3**	1E3	6E3
SPEAR-SWV-IBM	587	3600	0.97	**0.38**	0.39	0.39	0.38	98	**8.4**	59	102	1E3	**74**	153	1E3
SPEAR-IBM	1004	3600	1.18	**0.39**	0.42	0.42	0.38	130	**11**	78	136	1E3	**74**	153	1E3
SPEAR-SWV	60	3600	0.54	**0.36**	0.34	0.34	0.34	57	**4.9**	34	59	217	**17**	123	226
tnm-RANDSAT	568	3600	1.05	**0.88**	0.97	0.9	0.88	63	**26**	56	70	221	**35**	145	230
SAPS-RANDSAT	1019	3600	1	**0.67**	0.71	**0.65**	**0.66**	63	**26**	56	70	221	**35**	145	230
CPLEX-BIGMIX	719	3600	0.96	0.84	0.85	**0.63**	0.64	17	**0.13**	6.7	23	1E4	**6.6**	54	1E4
Gurobi-BIGMIX	992	3600	1.31	1.28	1.31	**1.19**	**1.17**	17	**0.13**	6.7	23	1E4	**6.6**	54	1E4
SCIP-BIGMIX	1153	3600	0.77	0.67	0.72	**0.58**	**0.57**	17	**0.13**	6.7	23	1E4	**6.6**	54	1E4
lp_solve-BIGMIX	3034	3600	0.58	**0.51**	0.53	**0.49**	**0.49**	17	**0.13**	6.7	23	1E4	**6.6**	54	1E4
CPLEX-CORLAT	430	3600	0.77	0.62	0.65	**0.47**	**0.47**	0.02	**0.01**	5.0	5.0	0.05	**0.03**	8.5	8.5
Gurobi-CORLAT	52	2159	0.6	0.48	0.5	**0.37**	0.38	0.02	**0.01**	5.0	5.0	0.05	**0.03**	8.5	8.5
SCIP-CORLAT	99	3600	0.59	0.47	0.48	**0.38**	**0.38**	0.02	**0.01**	5.0	5.0	0.05	**0.03**	8.5	8.5
lp_solve-CORLAT	2328	3600	0.57	0.52	0.54	0.43	**0.41**	0.02	**0.01**	5.0	5.0	0.05	**0.03**	8.5	8.5
CPLEX-RCW	364	3600	**0.02**	0.02	0.02	0.03	0.02	11	**3.3**	13	20	18	**3.5**	14	27
CPLEX-REG	402	3600	0.77	0.55	0.6	**0.42**	**0.42**	0.47	**0.02**	8.4	8.9	0.8	**0.05**	8.7	9.2
CPLEX-CR	416	3600	0.78	0.59	0.61	**0.45**	**0.44**	0.25	**0.02**	6.7	6.9	0.8	**0.05**	8.7	9.2
CPLEX-CRR	399	3600	0.64	0.48	0.51	**0.37**	**0.36**	3.7	**1.1**	8.7	11	18	**3.5**	14	27
LK-H-RUE	109	3600	0.69	0.71	**0.69**	**0.69**	0.67	6.0	**1.3**	18	49	9.9	**2.6**	60	97
LK-H-RCE	203	3600	0.82	0.81	0.81	0.8	**0.76**	5.7	**1.3**	12	129	9.3	**2.7**	27	235
LK-H-TSPLIB	429	3600	**1.01**	1.1	**0.84**	**0.94**	1.1	8.2	**1.8**	33	68	72	**16**	525	559
Concorde-RUE	490	3600	0.53	0.53	0.53	0.5	**0.45**	6.0	**1.3**	18	49	9.9	**2.6**	60	97
Concorde-RCE	118	3600	0.39	0.39	0.4	0.37	**0.35**	5.7	**1.3**	12	129	9.3	**2.7**	27	235
Concorde-TSPLIB	782	3600	**0.56**	**0.47**	**0.58**	**0.56**	**0.52**	8.2	**1.8**	33	68	72	**16**	525	559

Table 4

Quantitative evaluation of the impact of hyperparameter optimization on predictive accuracy. For each model family with hyperparameters, we report performance achieved with and without hyperparameter optimization (λ_{def} and λ_{opt}, respectively). We show 10-fold cross-validation performance for the default and for hyperparameters optimized using DIRECT with 2-fold cross-validation. For each dataset and model class, boldface denotes which of λ_{def} and λ_{opt} were not statistically significant from the better of the two (boldfacing 3E8 for RR and CPLEX-BIGMIX is not an error: its poor mean performance stems from a single outlier). Tables D.4 and D.5 (in the online appendix) provide results for all benchmarks.

| Domain	RMSE								Time to learn model (s)							
	RR		SP		NN		RF		RR		SP		NN		RF	
	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}
Minisat 2.0-COMPETITION	1.01	**0.93**	**1.25**	**1.12**	**0.62**	**0.61**	**0.47**	**0.47**	**6.8**	478	**28**	3.9E4	**22**	6717	**22**	631
SPEAR-INDU	**0.95**	**0.96**	**0.97**	29.5	**0.85**	**0.89**	**0.58**	**0.6**	**3.6**	212	**9.5**	6402	**5.4**	1069	**4.3**	139
CPLEX-BIGMIX	**3E8**	**0.91**	**0.93**	**0.93**	1.02	**0.91**	**0.64**	**0.64**	**3.4**	140	**8.3**	1257	**4.8**	213	**3.5**	111
Gurobi-CORLAT	**0.38**	**0.38**	0.44	**0.37**	**0.41**	**0.4**	**0.38**	**0.37**	**3.2**	254	**5.2**	1.0E4	**5.5**	408	**3.3**	101
LK-H-TSPLIB	9.55	**1.09**	**1.11**	**0.93**	1.77	**1.67**	1.06	**0.88**	**1.6**	50	**3.0**	406	**0.5**	57	**0.1**	5.0
Concorde-RUE	**0.41**	**0.41**	**0.43**	**0.42**	0.43	**0.41**	**0.45**	**0.44**	**4.2**	243	**3.6**	7362	**13**	574	**9.9**	283
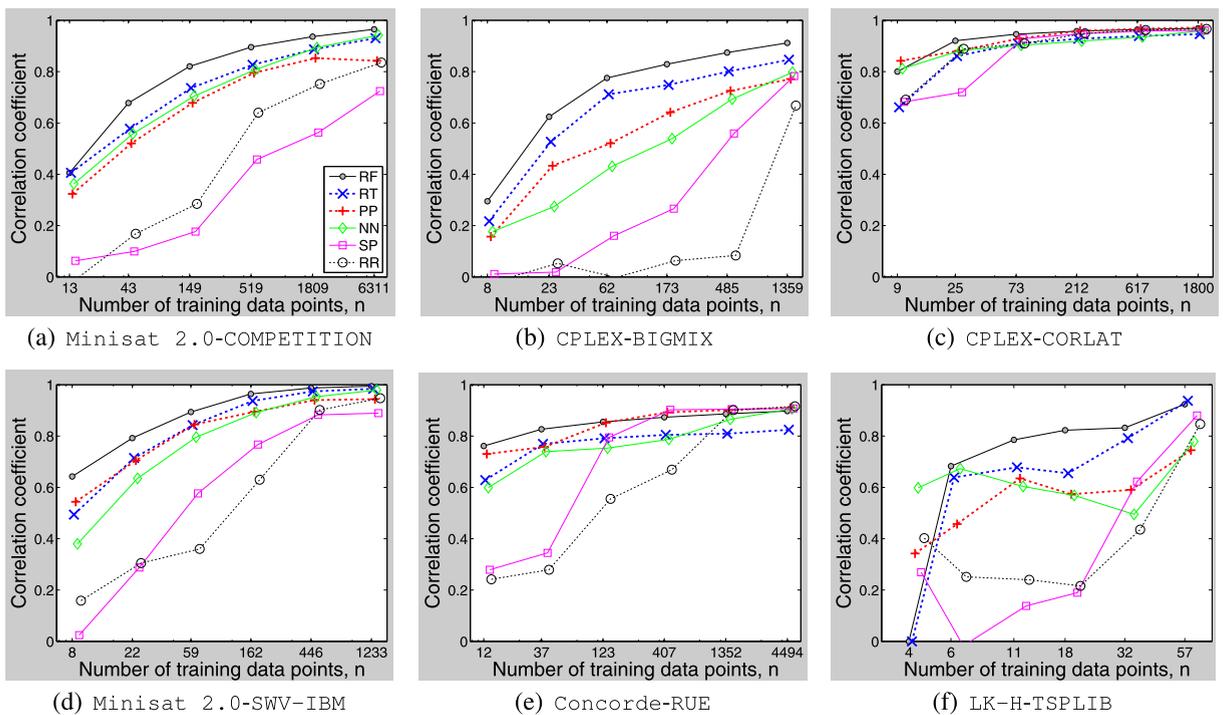
Fig. 5. Prediction quality for varying numbers of training instances. For each model and number of training instances, we plot the mean (taken across 10 cross-validation folds) correlation coefficient (CC) between true and predicted runtimes for new test instances; larger CC is better, 1 is perfect. Figs. D.11–D.13 (in the online appendix) show equivalent plots for the other benchmarks.

and in the following, we use CC rather than RMSE for such scaling plots, for two reasons. First, RMSE plots are often cluttered due to outlier instances for which prediction accuracy is poor (particularly for the ridge regression methods). Second, plotting CC facilitates performance comparisons across benchmarks, since CC $\in [-1, 1]$.

Overall, random forests performed best across training set sizes. Both versions of ridge regression (SP and RR) performed poorly for small training sets. This observation is significant, since most past work employed ridge regression with large amounts of data (*e.g.*, in SATzilla [119]), only measuring its performance in what turns out to be a favourable condition for it.

7. Performance predictions for new parameter configurations

We now move from predicting a single algorithm's runtime across a distribution of *instances* to predicting runtime across a family of algorithms (achieved by changing a given solver's *parameter settings* or *configurations*). For parameterized algorithms, there are four ways in which we can assess the prediction quality achieved by a model:

Table 5
Algorithms and characteristics of their parameter configuration spaces.

Algorithm	Parameter type	# parameters of this type	# values considered	Total # configurations
SPEAR	Categorical	10	2–20	
	Integer	4	5–8	8.34×10^{17}
	Continuous	12	3–6	
CPLEX	Boolean	6	2	
	Categorical	45	3–7	1.90×10^{47}
	Integer	18	5–7	
	Continuous	7	5–8	

1. **Predictions for training configurations on training instances.** Predictions for this most basic case are useful for succinctly modelling known algorithm performance data. Interestingly, several methods already perform poorly here.
2. **Predictions for training configurations on test instances.** Such predictions can be used to make a per-instance decision about which of a set of given parameter configurations will perform best on a previously unseen test instance, for example in algorithm selection [104,119,116,67].
3. **Predictions for test configurations on training instances.** This case is important in algorithm configuration, where the goal is to find high-quality parameter configurations for the given training instances [55,56].
4. **Predictions for test configurations on test instances.** This most general case is the most natural "pure prediction" problem (see [96,20]). It is also important for per-instance algorithm configuration, where one could use a model to search for the configuration that is most promising for a previously-unseen test instance [50].

We can understand the evaluation in the previous section as a special case of 2, where we only consider an algorithm's default configuration, but vary instances. We now consider the converse case 3, where instances do not vary, but configurations do. We consider case 4, in which we aim to generalize across both parameter configurations and instances, in Section 8.

7.1. Parameter configuration spaces

Here and in Section 8, we study two highly parameterized algorithms for two different problems: SPEAR for SAT and CPLEX for MIP.

For the industrial SAT solver SPEAR [3], we used the same parameter configuration space as in previous work [49]. This includes 26 parameters, out of which ten are categorical, four are integral, and twelve are continuous. The categorical parameters mainly control heuristics for variable and value selection, clause sorting, and resolution ordering, and also enable or disable optimizations, such as the pure literal rule. The continuous and integer parameters mainly deal with activity, decay, and elimination of variables and clauses, as well as with the randomized restart interval and percentage of random choices; we discretized each of them to between three and eight values. In total, and based on our discretization of continuous parameters, SPEAR has 8.34×10^{17} different configurations.

For the commercial MIP solver IBM ILOG CPLEX, we used the same configuration space with 76 parameters as in previous work [52]. These parameters exclude all CPLEX settings that change the problem formulation (*e.g.*, the optimality gap below which a solution is considered optimal). They include 12 preprocessing parameters (mostly categorical); 17 MIP strategy parameters (mostly categorical); 11 categorical parameters deciding how aggressively to use which types of cuts; 9 real-valued MIP "limit" parameters; 10 simplex parameters (half of them categorical); 6 barrier optimization parameters (mostly categorical); and 11 further parameters. In total, and based on our discretization of continuous parameters, these parameters gave rise to 1.90×10^{47} unique configurations.

7.2. Experimental setup

For the experiments in this and the next section, we gathered runtime data for SPEAR and CPLEX by executing each of them with 1000 randomly sampled parameter configurations. We ran each solver on instances from distributions for which we expected it to yield state-of-the-art performance: SPEAR on SWV and IBM; CPLEX on all MIP instance distributions discussed in the previous section. The runtime data for this and the next section was gathered on the 840-node Westgrid cluster Glacier (each of whose nodes is equipped with two 3.06 GHz Intel Xeon 32-bit processors and 2–4 GB RAM). Due to the large number of algorithm runs required for the experiments described in Section 8, we restricted the cutoff time of each single algorithm run to 300 seconds (compared to the 3000 seconds for the runs with the default parameter setting used in Section 6). In the following, we consider the performance of EPMs as parameters vary, but instance features do not; thus, here we used only one instance from each distribution and have no use for instance features. For each dataset, we selected the easiest benchmark instance amongst the ones for which the default parameter configuration required more than ten seconds on our reference machines. As before, we used 10-fold cross validation to assess the accuracy of our model predictions for previously unseen parameter configurations.

Table 6

Quantitative comparison of models for runtime predictions on previously unseen parameter configurations. We report 10-fold cross-validation performance. Lower RMSE is better (0 is optimal). Boldface indicates performance not statistically significantly different from the best method in each row. Table D.6 (in the online appendix) provides additional results (correlation coefficients and log likelihoods).

Domain	RMSE						Time to learn model (s)					
	RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
CPLEX-BIGMIX	0.26	0.34	0.38	**0.24**	0.33	**0.25**	5.33	**0.75**	3.66	34.26	4.24	2.98
CPLEX-CORLAT	**0.56**	0.67	0.78	**0.53**	0.75	**0.55**	5.36	**2.48**	3.88	32.53	4.19	**3**
CPLEX-REG	0.43	0.5	0.63	0.42	0.49	**0.38**	5.35	**2.09**	3.62	29.28	4	2.86
CPLEX-RCW	**0.2**	0.25	0.29	**0.21**	0.28	**0.21**	5.32	**0.43**	3.65	33.6	2.25	1.93
SPEAR-IBM	**0.25**	0.75	0.74	**0.25**	0.31	0.28	2.94	**0.17**	2.61	11.3	1.62	1.51
SPEAR-SWV	**0.36**	0.52	0.57	**0.35**	0.41	**0.36**	2.79	**0.14**	2.6	12.49	1.68	1.52
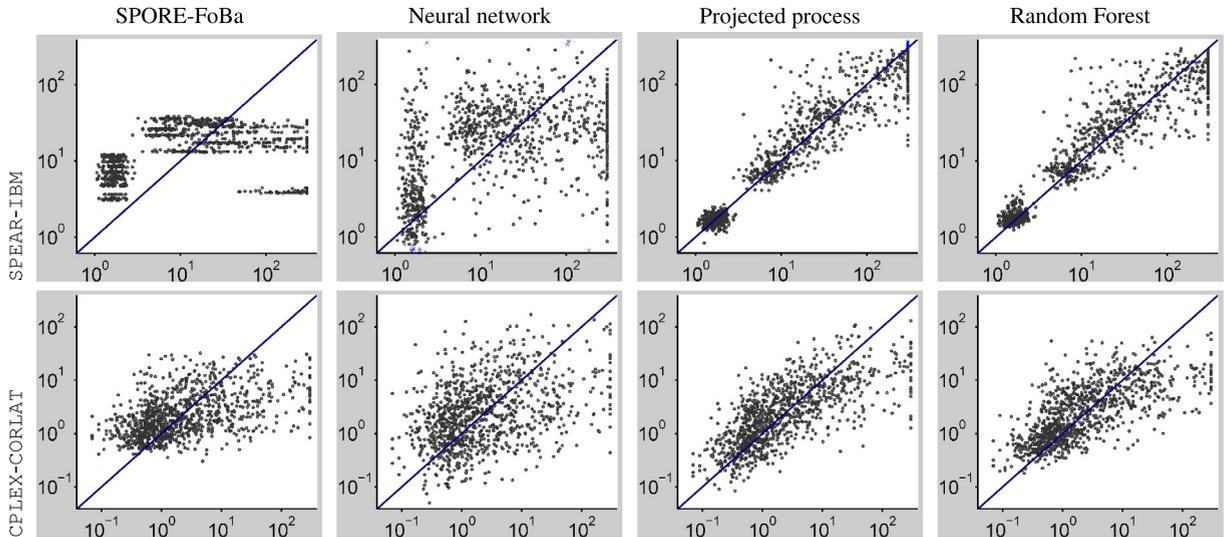
Fig. 6. Visual comparison of models for runtime predictions on previously unseen parameter configurations. In each scatter plot, the *x*-axis denotes true runtime and the *y*-axis cross-validated runtime as predicted by the respective model. Each dot represents one parameter configuration. Figs. D.14 and D.15 (in the online appendix) provide results for all domains and also show the performance of regression trees and ridge regression variant RR (whose predictions were similar to random forests and projected processes, with somewhat larger spread for regression trees).

7.3. Predictive quality

Table 6 quantifies the performance of all models on all benchmark problems, and Fig. 6 visualizes predictions. Again, we see that qualitatively, solver runtime as a function of parameter settings could be predicted quite well by most methods, even as runtimes varied by factors of over 1000 (see Fig. 6). We observe that projected processes, random forests, and ridge regression variant RR consistently outperformed regression trees; this is significant, as regression trees are the only model that has previously been used for predictions in configuration spaces with categorical parameters [8]. On the other hand, the poor performance of neural networks and of SPORE-FoBa (which mainly differs from variant RR in its feature expansion and selection) underlines that selecting the right (combinations of) features is not straightforward. Overall, the best performance was achieved by projected processes (applying our kernel function for categorical parameters from Section 4.1.2). As in the previous section, however, random forests were also either best or very close to the best for every data set.

7.4. Predictive quality with sparse training data

Results remained similar when varying the number of training configurations. As Fig. 7 shows, projected processes performed best overall, closely followed by random forests. Ridge regression variant RR often produced poor predictions when trained using a relatively small number of training data points, but performed well when given sufficient data. Finally, both SPORE-FoBa and neural networks performed relatively poorly regardless of the amount of data given.

8. Performance predictions in the joint space of instance features and parameter configurations

We now consider more challenging prediction problems for parameterized algorithms. In the first experiments discussed here (Sections 8.2 and 8.3) we tested predictions on the most challenging case, where both configurations and instances are
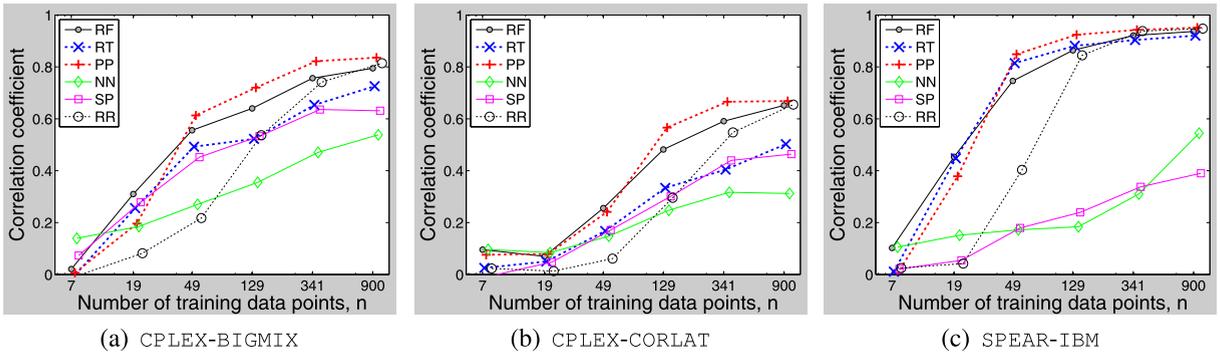
(a) CPLEX-BIGMIX (b) CPLEX-CORLAT (c) SPEAR-IBM

Fig. 7. Quality of predictions in the configuration space, as dependent on the number of training configurations. For each model and number of training instances, we plot mean $\pm$ standard deviation of the correlation coefficient (CC) between true and predicted runtimes for new test configurations. Fig. D.16 (in the online appendix) shows equivalent results for all benchmarks.

Table 7

Root mean squared error (RMSE) obtained by various models for runtime predictions on unseen instances and configurations. Boldface indicates the best average performance in each row. For CPLEX-BIGMIX, RR had a few extremely poorly predicted outliers, with the maximal prediction of $\log_{10}$ runtime exceeding 10^{100} (i.e., a runtime prediction above $10^{10^{100}}$); thus, we can only bound its RMSE from below. Models were based on 10 000 data points. Table D.7 (in the online appendix) provides additional results (correlation coefficients and log likelihoods).

Domain	RMSE						Time to learn model (s)					
	RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
CPLEX-BIGMIX	$> 10^{100}$	4.5	0.68	0.78	0.74	**0.55**	**25**	34	49	84	52	47
CPLEX-CORLAT	0.53	0.57	0.56	0.53	0.67	**0.49**	**26**	27	52	76	46	40
CPLEX-REG	**0.17**	0.19	0.19	0.19	0.24	**0.17**	23	**14**	50	77	32	31
CPLEX-RCW	0.1	0.12	0.12	0.12	0.12	**0.09**	24	**13**	45	78	25	24
CPLEX-CR	0.41	0.43	0.42	0.42	0.52	**0.38**	**26**	37	54	88	47	43
CPLEX-CRR	0.35	0.37	0.37	0.39	0.43	**0.32**	**29**	35	48	81	38	37
SPEAR-IBM	0.58	11	0.54	0.52	0.57	**0.44**	**15**	31	41	70	36	30
SPEAR-SWV	0.58	0.61	0.63	0.54	0.55	**0.44**	**15**	42	41	69	42	28
SPEAR-SWV-IBM	0.65	0.69	0.65	0.65	0.59	**0.45**	**17**	35	39	70	41	32

previously unseen. Later in the section (Section 8.4) we evaluate predictions made on all four combinations of training/test instances and training/test configurations.

8.1. Experimental setup

For the experiments in this section, we used SPEAR and CPLEX with the same configuration spaces as in Section 7 and the same $M = 1000$ randomly sampled configurations. We ran each of these configurations on all of the P problem instances in each of our instance sets (with P ranging from 604 to 2000), generating runtime data that can be thought of as a $M \times P$ matrix. We split both the M configurations and the P instances into training and test sets of equal size (using uniform random permutations). We then trained our EPMs on a fixed number of n randomly selected combinations of the $P/2$ training instances and $M/2$ training configurations.

We note that while a sound empirical evaluation of our methods required gathering a very large amount of data, such extensive experimentation is not required to use them in practice. The execution of the runs used in this section (between 604 000 and 2 000 000 per instance distribution) took over 60 CPU years, with time requirements for individual data sets ranging between 1.3 CPU years (SPEAR on SWV, where many runs took less than a second) and 18 CPU years (CPLEX on RCW, where most runs timed out). However, as we will demonstrate in Section 8.3, our methods often yield surprisingly accurate predictions based on data that can be gathered overnight on a single machine.

8.2. Predictive quality

We now examine the most interesting case, where test instances and configurations were both previously unseen. Table 7 provides quantitative results of model performance based on $n = 10\,000$ training data points, and Fig. 8 visualizes performance. Overall, we note that the best models generalized to new configurations *and* to new instances almost as well as to either alone (compare to Sections 6 and 7, respectively). On the most heterogeneous data set, CPLEX-BIGMIX, we once again witnessed extremely poorly predicted outliers for the ridge regression variants, but in all other cases, the models captured the large spread in runtimes (above 5 orders of magnitude) quite well. As in the experiments in Section 6.3, the tree-based approaches, which are able to model different regions of the input space independently, performed best on the
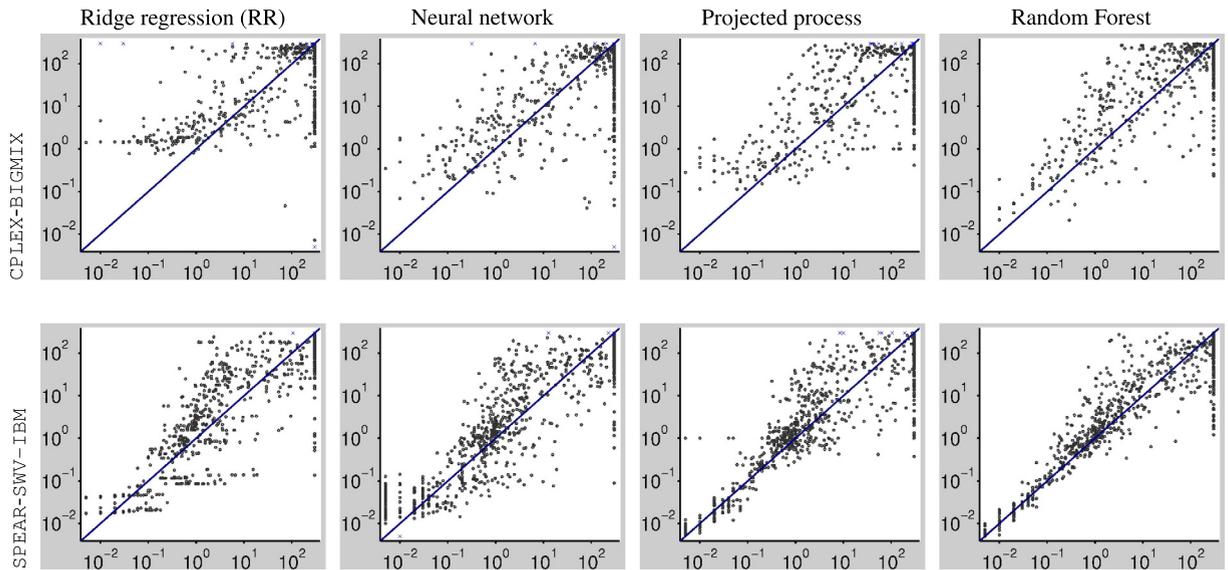
Fig. 8. Visual comparison of models for runtime predictions on pairs of previously unseen test configurations and instances. In each scatter plot, the *x*-axis shows true runtime and the *y*-axis cross-validated runtime as predicted by the respective model. Each dot represents one combination of an unseen instance and parameter configuration. Figs. D.17–D.19 (in the online appendix) include all domains and also show the performance of SPORE-FoBa (very similar to RR) and regression trees (similar to RF, somewhat larger spread).

most heterogeneous data sets. Fig. 8 also shows some qualitative differences in predictions: for example, ridge regression, neural networks, and projected processes sometimes overpredicted the runtime of the shortest runs, while the tree-based methods did not have this problem. Random forests performed best in all cases, which is consistent with their robust predictions in both the instance and the configuration space observed earlier.

8.3. Predictive quality with sparse training data

Next, we studied the amount of data that was actually needed to obtain good predictions, varying the number *n* of randomly selected combinations of training instances and configurations. Fig. 9 shows the correlation coefficients achieved by the various methods as a function of the amount of training data available. Overall, we note that most models already performed remarkably well (yielding correlation coefficients of 0.9 and higher) based on a few hundred training data points. This confirmed the practicality of our methods: on a single machine, it takes at most 12.5 hours to execute 150 algorithm runs with a cutoff time of 300 seconds. Thus, even users without access to a cluster can expect to be able to execute sufficiently many algorithm runs overnight to build a decent empirical performance model for their algorithm and instance distribution of interest. Examining our results in some more detail, the ridge regression variants again had trouble on the most heterogeneous benchmark CPLEX-BIGMIX, but otherwise performed quite well. Overall, random forests performed best across different training set sizes. Naturally, all methods required more data to make good predictions for heterogeneous benchmarks (*e.g.*, CPLEX-BIGMIX) than for relatively homogeneous ones (*e.g.*, CPLEX-CORLAT, for which the remarkably low number of 30 data points already yielded correlation coefficients exceeding 0.9).

8.4. Evaluating generalization performance in instance and configuration space

Now, we study all four combinations of predictions on training/test instances and training/test configurations. (See the beginning of Section 7 for a description of each scenario.) Our results are summarized in Table 8 and Figs. 10 and 11. For the figures, we sorted instances by average hardness (across configurations), and parameter configurations by average performance (across instances), generating a heatmap with instances on the *x*-axis, configurations on the *y*-axis, and greyscale values representing algorithm runtime for given configuration/instance combinations. We compare heatmaps representing true runtimes against those based on the predictions obtained from each of our models. Here, we only show results for the two scenarios where the performance advantage of random forests (the overall best method based on our results reported so far) over the other methods was highest (the heterogeneous data set SPEAR-SWV-IBM) and lowest (the homogeneous data set CPLEX-CORLAT); heatmaps for all data sets and model types are given in Figs. D.21–D.29 in the online appendix.

Fig. 10 shows the results for benchmark SPEAR-SWV-IBM. It features one column for each of the four combinations of training/test instances and training/test configurations, allowing us to visually assess how well the respective generalization works for each of the models. We note that in this case, the true heatmaps are almost indistinguishable from those predicted by random forests (and regression trees). Even for the most challenging case of unseen problem instances and
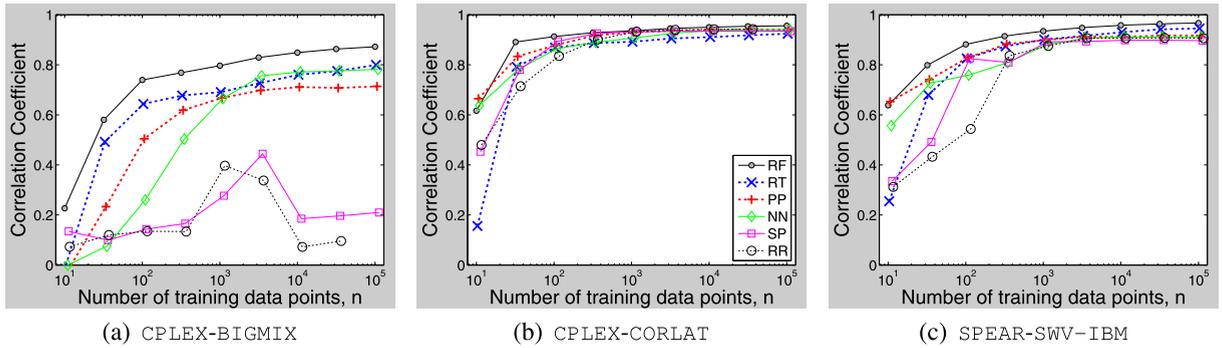
Fig. 9. Quality of predictions in the joint instance/configuration space as a function of the number of training data points. For each model and number of training data points, we plot mean correlation coefficients between true and predicted runtimes for new test instances and configurations. We omit standard deviations to avoid clutter, but they are very high for the two ridge regression variants. Figure D.20 (in the online appendix) shows corresponding, and qualitatively similar, results for all benchmarks.
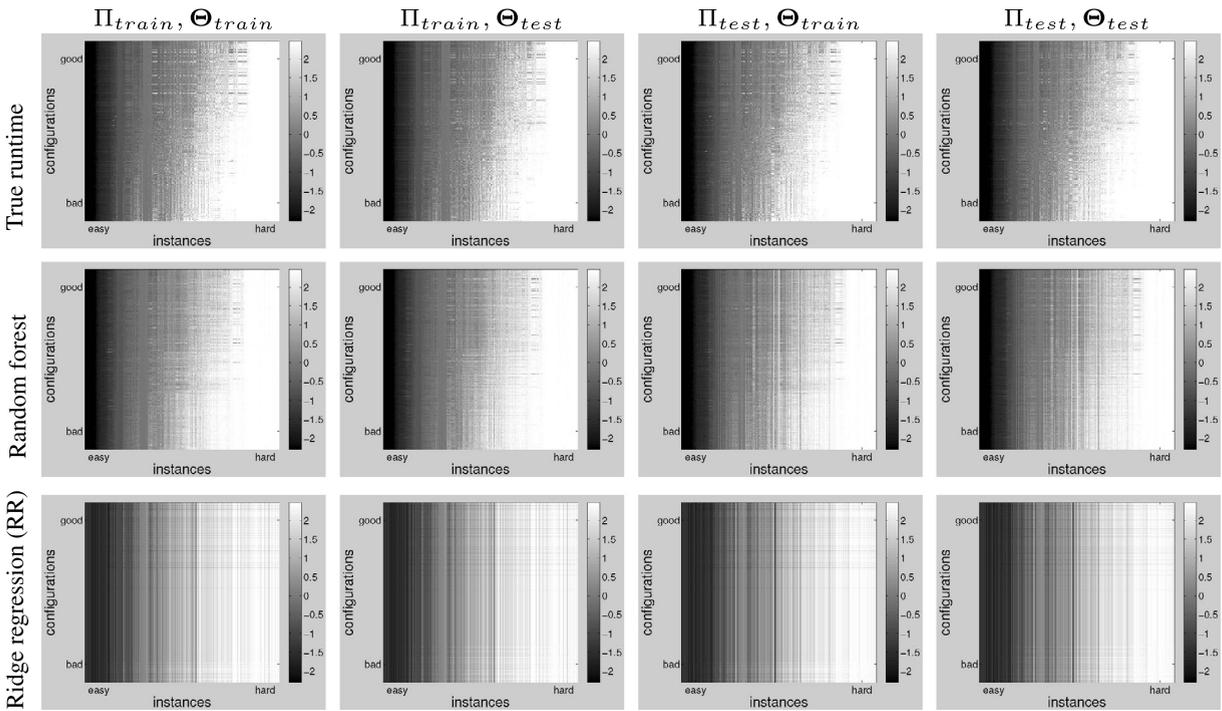

Fig. 10. True and predicted runtime matrices for dataset SPEAR-SWV-IBM, for all combinations of training/test instances (Π_{train} and Π_{test}, respectively) and training test configurations (Θ_{train} and Θ_{test}, respectively). For example, the top left heatmap shows the true runtimes for the cross product of 500 training configurations of SPEAR and the 684 training instances of the SWV-IBM benchmark set. Darker greyscale values represent faster runs, *i.e.*, instances on the right side of each heatmap are hard (they take longer to solve), and configurations at the top of each heapmap are good (they solve instances faster). (Plots for all models and benchmarks are given in Figs. D.21–D.29, in the online appendix.) The predicted matrix of regression trees (not shown) is visually indistinguishable from that of random forests, and those of all other methods closely resemble that of ridge regression.

parameter configurations, the tree-based methods captured the non-trivial interaction pattern between instances and parameter configurations. On the other hand, the non-tree-based methods (ridge regression variants, neural networks, and projected processes) only captured instance hardness, failing to distinguish good from bad configurations even in the simplest case of predictions for training instances and training configurations.

Fig. 11 shows the results for benchmark CPLEX-CORLAT. For the simplest case of predictions on training instances and configurations, the tree-based methods yielded predictions close to the true runtimes, capturing both instance hardness and performance of parameter configurations. In contrast, even in this simple case, the other methods only captured instance hardness, predicting all configurations to be roughly equal in performance. Random forests generalized better to test instances than to test configurations (compare the 3rd and 2nd columns of Fig. 11); this trend is also evident quantitatively in Table 8 for all CPLEX benchmarks. Regression tree predictions were visually indistinguishable from those of random forests;
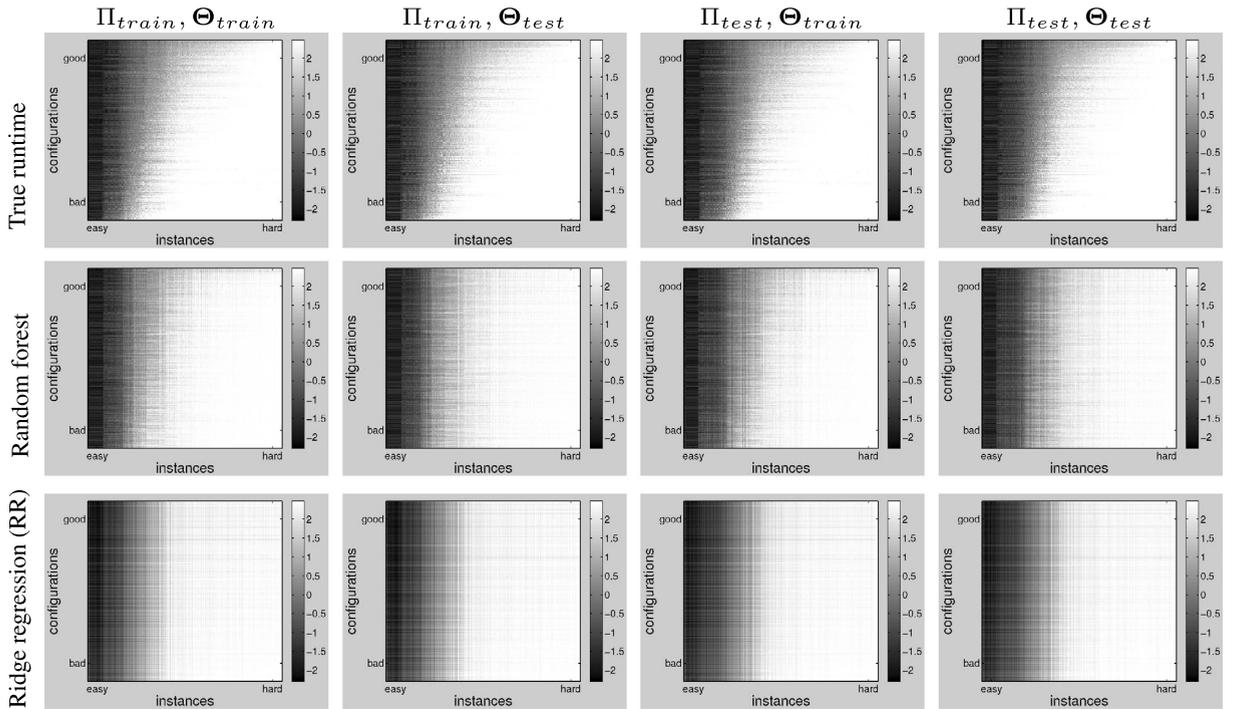
Fig. 11. Same type of data as in Fig. 10 but for dataset `CPLEX-CORLAT`.

Table 8
Root mean squared error (RMSE) obtained by various empirical performance models for predicting the runtime based on combinations of parameter configurations and instance features. We trained on 10 000 randomly-sampled combinations of training configurations and instances, and report performance for the four combinations of training/test instances and training/test configurations. Boldface indicates the model with the best performance.

Domain	Instances	Training configurations						Test configurations					
		RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
CPLEX-BIGMIX	Training	0.6	0.6	0.55	0.65	0.59	**0.43**	0.6	0.6	0.56	0.65	0.62	**0.45**
	Test	$> 10^{100}$	4.5	0.67	0.78	0.71	**0.54**	$> 10^{100}$	4.5	0.68	0.78	0.74	**0.55**
CPLEX-CORLAT	Training	0.5	0.55	0.47	0.49	0.54	**0.39**	0.52	0.56	0.54	0.51	0.64	**0.46**
	Test	0.51	0.55	0.5	0.51	0.58	**0.42**	0.53	0.57	0.56	0.53	0.67	**0.49**
CPLEX-REG	Training	0.15	0.18	0.15	0.16	0.17	**0.12**	**0.16**	0.18	0.18	0.17	0.22	**0.16**
	Test	0.17	0.19	0.17	0.18	0.19	**0.14**	**0.17**	0.19	0.19	0.19	0.24	**0.17**
CPLEX-RCW	Training	0.09	0.11	0.09	0.1	0.08	**0.06**	0.1	0.12	0.11	0.11	0.12	**0.09**
	Test	0.09	0.12	0.09	0.11	0.08	**0.06**	0.1	0.12	0.12	0.12	0.12	**0.09**
CPLEX-CR	Training	0.39	0.41	0.37	0.4	0.45	**0.32**	0.4	0.42	0.41	0.41	0.49	**0.36**
	Test	0.4	0.42	0.38	0.41	0.47	**0.34**	0.41	0.43	0.42	0.42	0.52	**0.38**
CPLEX-CRR	Training	0.33	0.35	0.33	0.36	0.38	**0.28**	0.34	0.36	0.36	0.37	0.41	**0.31**
	Test	0.34	0.37	0.34	0.38	0.4	**0.29**	0.35	0.37	0.37	0.39	0.43	**0.32**
SPEAR-IBM	Training	0.57	0.64	0.5	0.48	0.43	**0.34**	0.57	0.64	0.51	0.48	0.45	**0.36**
	Test	0.57	11	0.53	0.52	0.57	**0.42**	0.58	11	0.54	0.52	0.57	**0.44**
SPEAR-SWV	Training	0.52	0.56	0.56	0.46	0.37	**0.3**	0.52	0.56	0.57	0.47	0.43	**0.34**
	Test	0.57	0.61	0.62	0.53	0.51	**0.4**	0.58	0.61	0.63	0.54	0.55	**0.44**
SPEAR-SWV-IBM	Training	0.63	0.66	0.61	0.62	0.48	**0.36**	0.63	0.66	0.61	0.62	0.5	**0.38**
	Test	0.64	0.69	0.64	0.65	0.58	**0.43**	0.65	0.69	0.65	0.65	0.59	**0.45**

this strong qualitative performance is remarkable, considering that quantitatively they performed worse than other methods in terms of measures such as RMSE (see the results for `CPLEX-CORLAT` in Table 8).

Finally, we investigated once more how predictive quality depends on the quantity of training data, focusing on random forests (Fig. 12). For `SPEAR-SWV-IBM`, 100 training data points sufficed to obtain random forest models that captured the most salient features (*e.g.*, they correctly determined the simplicity of the roughly 20% easiest instances); more training data points gradually improved qualitative predictions, especially in distinguishing good from bad configurations. Likewise, for
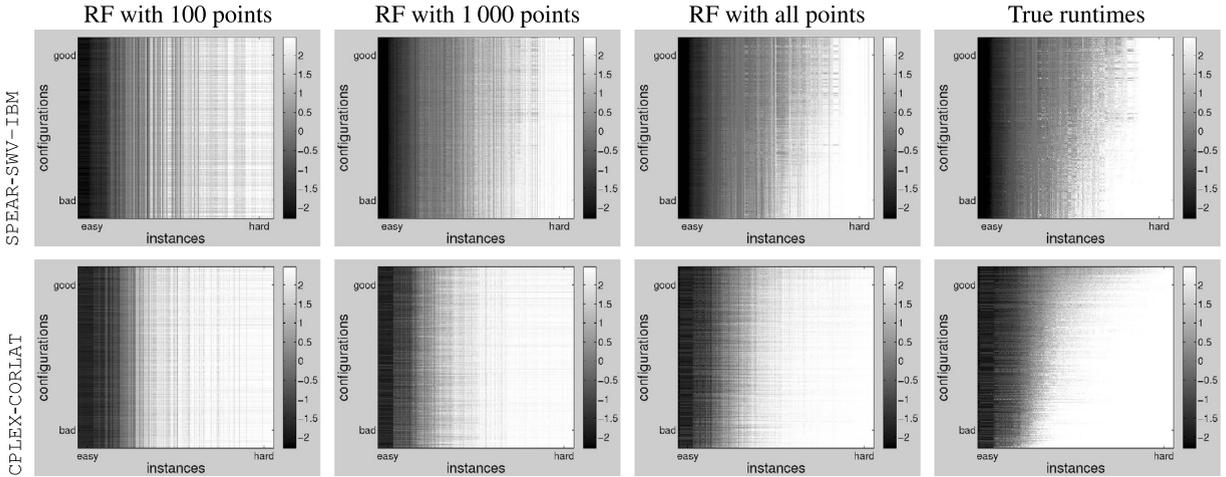
Fig. 12. Predicted runtime matrices with different number of training data points, compared to true runtime matrix. "All points" means the entire crossproduct of training instances and training configurations (342 500 data points for SPEAR-SWV-IBM and 500 000 for CPLEX-CORLAT). (Plots for all benchmarks are given in Fig. D.30 in the online appendix.)

CPLEX-CORLAT, salient features (*e.g.*, the simplicity of the roughly 25% easiest instances) could be detected based on 100 training data points, and more training data improved qualitative predictions to capture some of the differences between good and bad configurations. Overall, increases in the training set size yielded diminishing returns, and even predictions based on the entire cross-product of training instances and parameter configurations (i.e., between 151 000 and 500 000 runs) were not much different from those based on a subset of 10 000 samples (representing 2% to 6.6% of the entire training data).

9. Improved handling of censored runtimes in random forests

Most past work on predicting algorithm runtime has treated algorithm runs that were terminated prematurely at a so-called *captime* κ as if they finished at time κ. Thus, we adopted the same practice in the model comparisons we have described so far (using captimes of 3000 seconds for the runs in Section 6 and 300 seconds for the runs in Sections 7 and 8). Now, we revisit this issue for random forests.

Formally, terminating an algorithm run after a captime (or *censoring threshold*) κ yields a *right-censored* data point: we learn that κ is a lower bound on the actual time the algorithm run required. Let y_i denote the *actual* (unknown) runtime of algorithm run i. Under partial right censoring, our training data is $(\mathbf{x}_i, z_i, c_i)_{i=1}^n$, where $\mathbf{x}_i$ is our usual input vector (a vector of instance features, parameter values, or both combined), $z_i \in \mathbb{R}$ is a (possibly censored) runtime observation, and $c_i \in \{0, 1\}$ is a censoring indicator such that $z_i = y_i$ if $c_i = 0$ and $z_i < y_i$ if $c_i = 1$.

Observe that the typical, simplistic strategy for dealing with censored data produces biased models; intuitively, treating slow runs as though they were faster than they really were biases our training data downwards, and hence likewise biases predictions. Statisticians, mostly in the literature on so-called "survival analysis" from actuarial science, have developed strategies for building unbiased regression models based on censored data [86]. (Actuaries need to predict when people will die, given mortality data and the ages of people still living.) Gagliolo et al. [28,27] were the first to use techniques from this literature for runtime prediction. Specifically, they used a method for handling censored data in parameterized probabilistic models and employed the resulting models to construct dynamic algorithm portfolios. In the survival analysis literature, Schmee & Hahn [100] described an iterative procedure for handling censored data points in linear regression models. We employed this technique to improve the runtime predictions made by our portfolio-based algorithm selection method SATzilla [117]. While to the best of our knowledge, no other methods from this literature have been applied to algorithm runtime prediction, there exist several candidates for consideration in future work. In Gaussian processes, one could use approximations to handle the non-Gaussian observation likelihoods resulting from censorship; for example, Ertin [25] described a Laplace approximation for handling right-censored data. Random forests (RFs) have previously been adapted to handle censored data [102,44], but the classical methods yield non-parametric Kaplan–Meier estimators that are undefined beyond the largest uncensored data point. Here, we describe a simple improvement of the method by Schmee & Hahn [100] for use with random forests that we developed in the context of handling censored data in model-based algorithm configuration [54,51].

We denote the probability density function (PDF) and cumulative distribution function (CDF) of a Normal distribution by φ and Φ, respectively. Let $\mathbf{x}_i$ be an input with censored runtime κ_i. Given a Gaussian predictive distribution $\mathcal{N}(\mu_i, \sigma_i^2)$, the truncated Gaussian distribution $\mathcal{N}(\mu_i, \sigma_i^2)_{\geqslant \kappa_i}$ is defined by the PDF

$$p(y) = \begin{cases} 0 & y < \kappa_i \\ \frac{1}{\sigma_i} \varphi(\frac{x-\mu_i}{\sigma_i})/(1 - \Phi(\frac{\mu_i-\kappa_i}{\sigma_i})) & y \geqslant \kappa_i. \end{cases}$$

The method of Schmee & Hahn [100] is an Expectation Maximization (EM) algorithm. Applied to an RF model as its base model, that algorithm would first fit an initial RF using only uncensored data and then iterate between the following steps:

(E) For each tree T in the RF and each i s.t. $c_i = 1$: $\hat{y}_i^{(T)} \leftarrow$ mean of $\mathcal{N}(\mu_i, \sigma_i^2)_{\geqslant z_i}$;

(M) Refit the RF using $(\boldsymbol{\theta}_i, \hat{y}_i^{(T)})_{i=1}^n$ as the basis for tree T.

Here, $\mathcal{N}(\mu_i, \sigma_i^2)_{\geqslant z_i}$ denotes the predictive distribution of the current RF for data point i, *truncated at z_i, that is, conditioned on the fact that it is at least as large as z_i*. While the mean of $\mathcal{N}(\mu_i, \sigma_i^2)_{\geqslant z_i}$ is the best *single* value to impute for the ith data point, in the context of RF models this approach yields overly confident predictions: all trees would perfectly agree on the predictions for censored data points. To preserve our uncertainty about the true runtime of censored runs, we can change the E step to:

(E') For each tree T in the RF and each i s.t. $c_i = 1$: $\hat{y}_i^{(T)} \leftarrow$ sample from $\mathcal{N}(\mu_i, \sigma_i^2)_{\geqslant z_i}$.

In order to guarantee convergence, we also keep the assignment of bootstrap data points to each of the forest's trees fixed across iterations and draw the samples for each censored data point in a stratified manner; for brevity, we refer the reader to [51] for the precise details. Our resulting modified variant of Schmee & Hahn's algorithm takes our prior uncertainty into account when computing the posterior predictive distribution, thereby avoiding overly confident predictions. As an implementation detail, to avoid potentially large outlying predictions above the known maximal runtime of $\kappa_{max} = 300$ seconds, we ensure that the mean imputed value does not exceed κ_{max}.[11] (In the absence of censored runs—the case addressed in the major part of our work—this mechanism is not needed, since all predictions are linear combinations of observed runtimes and are thus upper-bounded by their maximum.)

9.1. Experimental setup

We now experimentally compare Schmee & Hahn's procedure and our modified version to two baselines: ignoring censored data points altogether and treating data points that were censored at the captime κ as uncensored data points with runtime κ. We only report results for the most interesting case of predictions for previously unseen parameter configurations and instances. We used the 9 benchmark distributions from Section 8, artificially censoring the training data at different thresholds below the actual threshold. We experimented with two different types of capped data: (1) data with a fixed censoring threshold across all data points, and (2) data in which the thresholds were instance-specific (specifically, we set the threshold for all runs on an instance to the runtime of the best of the 1000 configurations on that instance). The fixed threshold represents the sort of data generated by experimental studies like those from the previous sections of this paper, while the instance-specific threshold models practical applications of EPMs in model-based algorithm configuration procedures [54]. For both types of capped data and for all prediction strategies, we measured both predictive error (using RMSE as in the rest of the paper) and the quality of uncertainty estimates (using log likelihood, LL, as defined in Section 6.2) on the uncensored part of the test data.

9.2. Experimental results

Fig. 13 illustrates the raw predictions for one benchmark, demonstrating the qualitative differences between the four methods for treating capped data. In the case of a fixed censoring threshold κ, simply dropping censored data yielded consistent underestimates (see the top-left plot of Fig. 13), while treating censored data as uncensored at κ yielded good predictions up to κ (but not above); this strategy is thus reasonable when no predictions beyond κ are required (which is often the case; *e.g.*, throughout the main part of this article). The Schmee & Hahn variants performed similarly up to κ, but yielded unbiased predictions up to about two times κ. We note that a factor of two is not very much compared to the orders of magnitude variation we observe in our data. Much better predictions of larger runtimes can be achieved by using the instance-specific captimes discussed above (see the lower half of Fig. 13), and we thus advocate the use of such varying captimes in order to enable better scaling to larger captimes.

A quantitative analysis (described in Section D.4 of the online appendix) showed that in the fixed-threshold case, dropping censored data led to the worst prediction errors; treating censored data as uncensored improved results; and using the Schmee & Hahn variants further reduced prediction errors. However, with fixed thresholds, the Schmee & Hahn variants

---

[11] In Schmee & Hahn's algorithm, this simply means imputing $\min\{\kappa_{max}, \text{mean}(\mathcal{N}(\mu_i, \sigma_i^2)_{\geqslant z_i})\}$. In our sampling version, it amounts to keeping track of the mean m_i of the imputed samples for each censored data point i and subtracting $m_i - \kappa_{max}$ from each sample for data point i if $m_i > \kappa_{max}$.
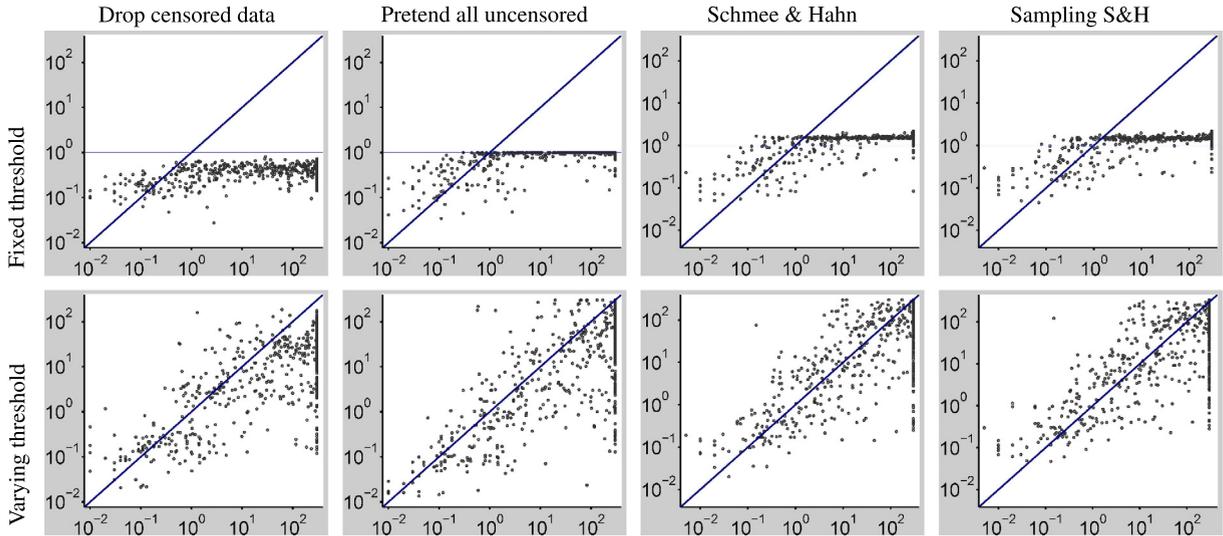
Fig. 13. True and predicted runtime of various ways of handling censored data in random forests, for scenario `CPLEX-BIGMIX` with fixed censoring threshold of one second during training (top) and varying threshold (bottom). In each scatterplot, the *x*-axis indicates true runtime and the *y*-axis cross-validated runtime as predicted by the respective model. Each dot represents one instance. Analogous figures for all benchmarks are given in Figs. D.31–D.39 (in the online appendix).

often yielded poor uncertainty estimates because they imputed similar values (close to the fixed threshold) for all censored data points, yielding too little variation across trees, and thus also yielded overconfident predictions. In contrast, for data with varying captimes, treating censored data as uncensored often performed worse than simply dropping it, and the Schmee & Hahn variants (in particular our new one) yielded both competitive uncertainty estimates and the lowest prediction error. Finally, we found these qualitative findings to be robust with respect to how aggressively (*i.e.*, how low) the captimes were chosen. Overall, random forests handled censored data reasonably well. We note that other models might be better suited to *extrapolating* from training data with short captimes to obtain accurate runtime predictions for long algorithm runs.

10. Conclusions

In this article, we assessed and advanced the state of the art in predicting the performance of algorithms for hard combinatorial problems. We proposed new techniques for building predictive models, with a particular focus on improving prediction accuracy for parameterized algorithms, and also introduced a wealth of new features for three of the most widely studied NP-hard problems (SAT, MIP and TSP) that benefit all models. We conducted the largest experimental study of which we are aware—predicting the performance of 11 algorithms on 35 instance distributions from SAT, MIP and TSP—comparing our new modelling approaches with a comprehensive set of methods from the literature. We showed that our new approaches—chiefly those based on random forests, but also approximate Gaussian processes—offer the best performance, whether we consider predictions for previously unseen problem instances for parameterless algorithms, new parameter settings for a parameterized algorithm running on a single problem instance, or parameterized algorithms being run both with new parameter values and on previously unseen problem instances. We also demonstrated in each of these settings that very accurate predictions (correlation coefficients between predicted and true runtime exceeding 0.9) are possible based on very small amounts of training data (only hundreds of runtime observations). Finally, we demonstrated how our best-performing model, random forests, could be improved further by better handling data from prematurely terminated runs. Overall, we showed that our methods are fast, general, and achieve good, robust performance. We hope they will be useful to a wide variety of researchers who seek to model algorithm performance for algorithm analysis, scheduling, algorithm portfolio construction, automated algorithm configuration, and other applications. The Matlab source code for our models, the data and source code to reproduce our experiments, and an online appendix containing additional experimental results, are available online at http://www.cs.ubc.ca/labs/beta/Projects/EPMs.

Appendix A. Details on benchmark instance sets

This appendix gives more information about our instance benchmarks. For the SAT benchmarks, the number of variables and clauses are given for the original instance (before preprocessing). (In contrast, [48] reported these numbers after preprocessing, explaining the differences in reported values for IBM and SWV.)

A.1. SAT benchmarks

INDU This benchmark data set comprises 1676 instances from the industrial categories of the 2002–2009 SAT competitions as well as from the 2006, 2008 and 2010 SAT Races. These instances contain an average of 111 000 variables and 689 187 clauses, with respective standard deviations of 318 955 and 1 510 764, and respective maxima of 9 685 434 variables and 14 586 886 clauses.

HAND This benchmark data set comprises 1955 instances from the handmade categories of the 2002–2009 SAT Competitions. These instances contain an average of 4968 variables and 82 594 clauses, with respective standard deviations of 21 312 and 337 760, and respective maxima of 270 000 variables and 4 333 038 clauses.

RAND This benchmark data set comprises 3381 instances from the random categories of the 2002–2009 SAT Competitions. These instances contain an average of 1048 variables and 6626 clauses, with respective standard deviations of 2593 and 11 221, and respective maxima of 19 000 variables and 79 800 clauses.

COMPETITION This set is the union of INDU, HAND, and RAND.

IBM This set of SAT-encoded bounded model checking instances comprises 765 instances generated by Zarpas [122]; these instances were selected as the instances in 40 randomly-selected folders from the IBM Formal Verification Benchmarks Library. These instances contained an average of 96 454 variables and 413 143 clauses, with respective standard deviations of 169 859 and 717 379, and respective maxima of 1 621 756 variables and 6 359 302 clauses.

SWV This set of SAT-encoded software verification instances comprises 604 instances generated with the CALYSTO static checker [4], used for the verification of five programs: the spam filter Dspam, the SAT solver HyperSAT, the Wine Windows OS emulator, the gzip archiver, and a component of xinetd (a secure version of inetd). These instances contain an average of 68 935 variables and 206 147 clauses, with respective standard deviations of 56 966 and 181 714, and respective maxima of 280 972 variables and 926 872 clauses.

RANDSAT This set contains 2076 satisfiable instances (proved by at least one winning solver from the previous SAT competitions) from data set RAND. These instances contain an average of 1380 variables and 8042 clauses, with respective standard deviations of 3164 and 13 434, and respective maxima of 19 000 variables and 79 800 clauses.

A.2. MIP benchmarks

BIGMIX This highly heterogeneous mix of publicly available Mixed Integer Linear Programming (MILP) benchmarks comprises 1510 MILP instances. The instances in this set have an average of 8610 variables and 4250 constraints, with respective standard deviations of 34 832 and 21 009, and respective maxima of 550 539 variables and 550 339 constraints.

CORLAT This set comprises 2000 MILP instances based on real data used for the construction of a wildlife corridor for grizzly bears in the Northern Rockies region (the instances were described by Gomes et al. [32] and made available to us by Bistra Dilkina). All instances had 466 variables; on average they had 486 constraints (with standard deviation 25.2 and a maximum of 551).

RCW This set comprises 1980 MILP instances from a computational sustainability project. These instances model the spread of the endangered red-cockaded woodpecker, conditional on decisions about certain parcels of land to be protected. We generated 1980 instances (20 random instances for each combination of 9 maps and 11 budgets), using the generator from [1] with the same parameter setting as used in that paper, except a smaller sample size of 5. All instances have 82 346 variables; on average, they have 328 816 constraints (with a standard deviation of only 3 and a maximum of 328 820).

REG This set comprises 2000 MILP-encoded instances of the winner determination problem in combinatorial auctions. We generated 2000 instances using the regions generator from the Combinatorial Auction Test Suite [77], with the number of bids selected uniformly at random from between 750 and 1250, and a fixed bids/goods ratio of 3.91 (following [76]). They have an average of 1129 variables and 498 constraints, with respective standard deviations of 73 and 32 and respective maxima of 1255 variables and 557 constraints.

A.3. TSP benchmarks

RUE This set comprises 4993 uniform random Euclidean 2-dimensional TSP instances generated by the random TSP generator, `portgen` [63]. The number of nodes was randomly selected from 100 to 1600, and the generated TSP instances contain an average of 849 nodes with a standard deviation of 429 and a maximum of 1599 nodes.

RCE This set comprises 5001 random clustered Euclidean 2-dimensional TSP instances generated by the random TSP generator, portcgen [63]. The number of nodes was randomly selected from 100 to 1600, and the number of clusters was set to 1% of the number of nodes. The generated TSP instances contain an average of 852 nodes with a standard deviation of 432 and a maximum of 1599 nodes.

TSPLIB This set contains a subset of the prominent TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/) repository. We only included the 63 instances for which both our own feature computation code and the code by Smith-Miles & van Hemert [107] completed successfully (ours succeeded on 23 additional instances). These 63 instances have 931 ± 1376 nodes, with a range from 100 to 5934.

References

[1] K. Ahmadizadeh, B. Dilkina, C. Gomes, A. Sabharwal, An empirical study of optimization for maximizing diffusion in networks, in: Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10), in: LNCS, vol. 6308, Springer-Verlag, 2010, pp. 514–521.

[2] D.L. Applegate, R.E. Bixby, V. Chvátal, W.J. Cook, The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2006.

[3] D. Babić, Exploiting structure for scalable software verification, PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[4] D. Babić, A.J. Hu, Structural abstraction of software verification conditions, in: Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), in: LNCS, vol. 4590, Springer-Verlag, 2007, pp. 366–378.

[5] D. Babić, F. Hutter, Spear theorem prover. Solver description, SAT competition 2007, 2007.

[6] T. Bartz-Beielstein, Experimental Research in Evolutionary Computation: The New Experimentalism, Natural Computing Series, Springer, 2006.

[7] T. Bartz-Beielstein, C. Lasarczyk, M. Preuss, Sequential parameter optimization, in: Proceedings of the 2004 Congress on Evolutionary Computation (CEC'05), 2005, pp. 773–780.

[8] T. Bartz-Beielstein, S. Markon, Tuning search algorithms for real-world applications: a regression tree based approach, in: Proceedings of the 2004 Congress on Evolutionary Computation (CEC'04), 2004, pp. 1111–1118.

[9] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, J. Mach. Learn. Res. 13 (2012) 281–305.

[10] M. Berkelaar, J. Dirks, K. Eikland, P. Notebaert, J. Ebert, lp_solve 5.5, http://lpsolve.sourceforge.net/5.5/index.htm, 2012. Last accessed on August 6, 2012.

[11] T. Berthold, G. Gamrath, S. Heinz, M. Pfetsch, S. Vigerske, K. Wolter, SCIP 1.2.1.4, http://scip.zib.de/doc/html/index.shtml, 2012. Last accessed on August 6, 2012.

[12] C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[13] G.E.P. Box, N. Draper, Response Surfaces, Mixtures, and Ridge Analyses, second edition, Wiley, 2007.

[14] G.E.P. Box, K. Wilson, On the experimental attainment of optimum conditions (with discussion), J. R. Stat. Soc. B 13 (1) (1951) 1–45.

[15] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.

[16] L. Breiman, J.H. Friedman, R. Olshen, C.J. Stone, Classification and Regression Trees, Wadsworth, 1984.

[17] E.A. Brewer, Portable high-performance supercomputing: high-level platform-dependent optimization, PhD thesis, Massachusetts Institute of Technology, 1994.

[18] E.A. Brewer, High-level optimization via automated statistical modeling, in: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP-95), 1995, pp. 80–91.

[19] P. Cheeseman, B. Kanefsky, W.M. Taylor, Where the really hard problems are, in: Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91), 1991, pp. 331–337.

[20] M. Chiarandini, Y. Goegebeur, Mixed models for the analysis of optimization algorithms, in: T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (Eds.), Experimental Methods for the Analysis of Optimization Algorithms, Springer-Verlag, 2010, pp. 225–264.

[21] W. Cook, Applications of the TSP, http://www.tsp.gatech.edu/apps/index.html, 2012. Last accessed on April 10, 2012.

[22] W. Cook, Concorde downloads page, http://www.tsp.gatech.edu/concorde/downloads/downloads.htm, 2012. Last accessed on October 24, 2012.

[23] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04), in: LNCS, vol. 3569, Springer-Verlag, 2005, pp. 61–75.

[24] N. Eén, N. Sörensson, An extensible SAT-solver, in: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03), 2004, pp. 502–518.

[25] E. Ertin, Gaussian process models for censored sensor readings, in: Proceedings of the IEEE Statistical Signal Processing Workshop 2007 (SSP'07), 2007, pp. 665–669.

[26] E. Fink, How to solve it automatically: Selection among problem-solving methods, in: Proceedings of the Fourth International Conference on AI Planning Systems, AAAI Press, 1998, pp. 128–136.

[27] M. Gagliolo, C. Legrand, Algorithm survival analysis, in: T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (Eds.), Experimental Methods for the Analysis of Optimization Algorithms, Springer, 2010, pp. 161–184.

[28] M. Gagliolo, J. Schmidhuber, Dynamic algorithm portfolios, in: International Symposium on Artificial Intelligence and Mathematics (ISAIM'06), 2006.

[29] C. Gebruers, A. Guerri, B. Hnich, M. Milano, Making choices using structure at the instance level within a case based reasoning framework, in: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04), in: LNCS, vol. 3011, Springer-Verlag, 2004, pp. 380–386.

[30] C. Gebruers, B. Hnich, D. Bridge, E. Freuder, Using CBR to select solution strategies in constraint programming, in: Proceedings of the 6th International Conference on Case Based Reasoning (ICCBR'05), in: LNCS, vol. 3620, Springer-Verlag, 2005, pp. 222–236.

[31] C.P. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, J. Autom. Reason. 24 (1) (2000) 67–100.

[32] C.P. Gomes, W.-J. van Hoeve, A. Sabharwal, Connections in networks: a hybrid approach, in: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'08), in: LNCS, vol. 5015, Springer-Verlag, 2008, pp. 303–307.

[33] A. Guerri, M. Milano, Learning techniques for automatic algorithm portfolio selection, in: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04), 2004, pp. 475–479.

[34] H. Guo, W.H. Hsu, A learning-based algorithm selection meta-reasoner for the real-time MPE problem, in: Proceedings of the 17th Australian Conference on Artificial Intelligence (AI'04), in: LNCS, vol. 3339, Springer-Verlag, 2004, pp. 307–318.

[35] Gurobi Optimization Inc. Gurobi 2.0., http://www.gurobi.com/. Last accessed on August 6, 2012.

[36] I. Guyon, S. Gunn, M. Nikravesh, L. Zadeh, Feature Extraction, Foundations and Applications, Springer, 2006.

[37] S. Haim, T. Walsh, Online estimation of SAT solving runtime, in: Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08), in: LNCS, vol. 4996, Springer-Verlag, 2008, pp. 133–138.

[38] E.A. Hansen, S. Zilberstein, Monitoring the progress of anytime problem-solving, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence, Portland, Oregon, 1996, pp. 1229–1234.

[39] T. Hastie, R. Tibshirani, J.H. Friedman, The Elements of Statistical Learning, second edition, Springer Series in Statistics, Springer, 2009.

[40] K. Helsgaun, An effective implementation of the Lin–Kernighan traveling salesman heuristic, Eur. J. Oper. Res. 126 (1) (2000) 106–130.

[41] P. Herwig, Using graphs to get a better insight into satisfiability problems, Master's thesis, Delft University of Technology, Department of Electrical Engineering, Mathematics and Computer Science, 2006.

[42] H.H. Hoos, T. Stützle, Stochastic Local Search – Foundations & Applications, Morgan Kaufmann Publishers, 2005.

[43] E. Horvitz, Y. Ruan, C.P. Gomes, H. Kautz, B. Selman, D.M. Chickering, A Bayesian approach to tackling hard computational problems, in: Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI'01), 2001, pp. 235–244.

[44] T. Hothorn, B. Lausen, A. Benner, M. Radespiel-Tröger, Bagging survival trees, Stat. Med. 23 (2004) 77–91.

[45] A.E. Howe, E. Dahlman, C. Hansen, M. Scheetz, A. Mayrhauser, Exploiting competitive planner performance, in: S. Biundo, M. Fox (Eds.), Recent Advances in AI Planning (ECP'99), in: Lecture Notes in Computer Science, vol. 1809, Springer, Berlin, Heidelberg, 2000, pp. 62–72.

[46] E.I. Hsu, C. Muise, J.C. Beck, S.A. McIlraith, Probabilistically estimating backbones and variable bias: experimental overview, in: Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08), in: LNCS, vol. 5202, Springer-Verlag, 2008, pp. 613–617.

[47] L. Huang, J. Jia, B. Yu, B. Chun, P. Maniatis, M. Naik, Predicting execution time of computer programs using sparse polynomial regression, in: Proceedings of the 23rd Conference on Advances in Neural Information Processing Systems (NIPS'10), 2010, pp. 883–891.

[48] F. Hutter, Automated configuration of algorithms for solving hard computational problems, PhD thesis, University of British Columbia, Department of Computer Science, Vancouver, Canada, 2009.

[49] F. Hutter, D. Babić, H.H. Hoos, A.J. Hu, Boosting verification by automatic tuning of decision procedures, in: Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07), 2007, pp. 27–34.

[50] F. Hutter, Y. Hamadi, H.H. Hoos, K. Leyton-Brown, Performance prediction and automated tuning of randomized and parametric algorithms, in: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06), in: LNCS, vol. 4204, Springer-Verlag, 2006, pp. 213–228.

[51] F. Hutter, H. Hoos, K. Leyton-Brown, Bayesian optimization with censored response data, ArXiv e-prints arXiv:1310.1947 [cs.AI], 2013.

[52] F. Hutter, H.H. Hoos, K. Leyton-Brown, Automated configuration of mixed integer programming solvers, in: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'10), in: LNCS, vol. 6140, Springer-Verlag, 2010, pp. 186–202.

[53] F. Hutter, H.H. Hoos, K. Leyton-Brown, Tradeoffs in the empirical evaluation of competing algorithm designs, in: Special Issue on Learning and Intelligent Optimization, Ann. Math. Artif. Intell. 60 (1) (2010) 65–89.

[54] F. Hutter, H.H. Hoos, K. Leyton-Brown, Bayesian optimization with censored response data, in: NIPS 2011 Workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits, 2011, published online.

[55] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: Proceedings of the 5th Workshop on Learning and Intelligent Optimization (LION'11), in: LNCS, vol. 6683, Springer-Verlag, 2011, pp. 507–523.

[56] F. Hutter, H.H. Hoos, K. Leyton-Brown, Parallel algorithm configuration, in: Proceedings of the 6th Workshop on Learning and Intelligent Optimization (LION'12), in: LNCS, Springer-Verlag, 2012, pp. 55–70, http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-44972-7.

[57] F. Hutter, H.H. Hoos, K. Leyton-Brown, Identifying key algorithm parameters and instance features using forward selection, in: Proc. of LION-7, 2013, in press.

[58] F. Hutter, H.H. Hoos, K. Leyton-Brown, K.P. Murphy, An experimental investigation of model-based parameter optimisation: SPO and beyond, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'09), 2009, pp. 271–278.

[59] F. Hutter, H.H. Hoos, K. Leyton-Brown, K.P. Murphy, Time-bounded sequential parameter optimization, in: Proceedings of the 4th Workshop on Learning and Intelligent Optimization (LION'10), in: LNCS, vol. 6073, Springer-Verlag, 2010, pp. 281–298.

[60] F. Hutter, D.A.D. Tompkins, H.H. Hoos, Scaling and probabilistic smoothing: efficient dynamic local search for SAT, in: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02), in: LNCS, vol. 2470, Springer-Verlag, 2002, pp. 233–248.

[61] International Business Machines Corp, IBM ILOG CPLEX Optimizer – Data Sheet, ftp://public.dhe.ibm.com/common/ssi/ecm/en/wsd14044usen/WSD14044USEN.PDF, 2012. Last accessed on August 6, 2012.

[62] International Business, Machines Corp, CPLEX 12.1. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, 2012. Last accessed on August 6, 2012.

[63] D.S. Johnson, Random TSP generators for the DIMACS TSP challenge. http://www2.research.att.com/~dsj/chtsp/codes.tar, 2011. Last accessed on May 16, 2011.

[64] D.R. Jones, C.D. Perttunen, B.E. Stuckman, Lipschitzian optimization without the Lipschitz constant, J. Optim. Theory Appl. 79 (1) (1993) 157–181.

[65] D.R. Jones, M. Schonlau, W.J. Welch, Efficient global optimization of expensive black box functions, J. Glob. Optim. 13 (1998) 455–492.

[66] T. Jones, S. Forrest, Fitness distance correlation as a measure of problem difficulty for genetic algorithms, in: Proceedings of the 6th International Conference on Genetic Algorithms (ICGA'95), 1995, pp. 184–192.

[67] S. Kadioglu, Y. Malitsky, M. Sellmann, K. Tierney, ISAC – instance specific algorithm configuration, in: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI'10), 2010, pp. 751–756.

[68] P. Kilby, J. Slaney, S. Thiebaux, T. Walsh, Estimating search tree size, in: Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06), 2006, pp. 1014–1019.

[69] D. Knuth, Estimating the efficiency of backtrack programs, Math. Comput. 29 (129) (1975) 121–136.

[70] L. Kotthoff, I.P. Gent, I. Miguel, An evaluation of machine learning in algorithm selection for search problems, AI Commun. 25 (3) (2012) 257–270.

[71] D.G. Krige, A statistical approach to some basic mine valuation problems on the Witwatersrand, J. Chem. Metall. Min. Soc. S. Afr. 52 (6) (1951) 119–139.

[72] N.D. Lawrence, M. Seeger, R. Herbrich, Fast sparse Gaussian process methods: the informative vector machine, in: Proceedings of the 15th Conference on Advances in Neural Information Processing Systems (NIPS'02), 2003, pp. 609–616.

[73] K. Leyton-Brown, H.H. Hoos, F. Hutter, L. Xu, Understanding the empirical hardness of NP-complete problems, Commun. ACM (2013), in press.

[74] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham, Boosting as a metaphor for algorithm design, in: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03), in: LNCS, vol. 2833, Springer-Verlag, 2003, pp. 899–903.

[75] K. Leyton-Brown, E. Nudelman, Y. Shoham, Learning the empirical hardness of optimization problems: the case of combinatorial auctions, in: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02), in: LNCS, vol. 2470, Springer-Verlag, 2002, pp. 556–572.

[76] K. Leyton-Brown, E. Nudelman, Y. Shoham, Empirical hardness models: methodology and a case study on combinatorial auctions, J. ACM 56 (4) (2009) 1–52.

[77] K. Leyton-Brown, M. Pearson, Y. Shoham, Towards a universal test suite for combinatorial auction algorithms, in: EC '00: Proceedings of the 2nd ACM Conference on Electronic Commerce, 2000, pp. 66–76.

[78] S. Lin, B.W. Kernighan, An effective heuristic algorithm for the traveling-salesman problem, Oper. Res. 21 (2) (1973) 498–516.

[79] L. Lobjois, M. Lemaître, Branch and bound algorithm selection by performance prediction, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98), 1998, pp. 353–358.

[80] Y.S. Mahajan, Z. Fu, S. Malik, Zchaff2004: an efficient SAT solver, in: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05), in: LNCS, vol. 3542, Springer-Verlag, 2005, pp. 360–375.

[81] N. Meinshausen, Quantile regression forests, J. Mach. Learn. Res. 7 (2006) 983–999.

[82] O. Mersmann, B. Bischl, H. Trautmann, M. Wagner, J. Bossek, F. Neumann, A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem, Ann. Math. Artif. Intell. (2013), http://dx.doi.org/10.1007/s10472-013-9341-2, 32 p., published online: 28 March 2013.

[83] D. Mitchell, B. Selman, H. Levesque, Hard and easy distributions of SAT problems, in: Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92), 1992, pp. 459–465.

[84] I.T. Nabney, NETLAB: Algorithms for Pattern Recognition, Springer, 2002.

[85] V. Nannen, A.E. Eiben, Relevance estimation and value calibration of evolutionary algorithm parameters, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07), 2007, pp. 975–980.

[86] W. Nelson, Applied Life Data Analysis, Wiley Series in Probability and Statistics, John Wiley & Sons, 2003.

[87] J. Nocedal, S.J. Wright, Numerical Optimization, second edition, Springer, 2006.

[88] G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, D.V. Wilcox, Pace–a toolset for the performance prediction of parallel and distributed systems, Int. J. High Perform. Comput. Appl. 14 (3) (2000) 228–251.

[89] E. Nudelman, K. Leyton-Brown, G. Andrew, C. Gomes, J. McFadden, B. Selman, Y. Shoham, Satzilla 0.9. Solver description, 2003 SAT Competition, 2003.

[90] E. Nudelman, K. Leyton-Brown, H.H. Hoos, A. Devkar, Y. Shoham, Understanding random SAT: beyond the clauses-to-variables ratio, in: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04), in: LNCS, vol. 3258, Springer-Verlag, 2004, pp. 438–452.

[91] B. Pfahringer, H. Bensusan, C. Giraud-Carrier, Meta-learning by landmarking various learning algorithms, in: Proceedings of the 17th International Conference on Machine Learning (ICML'00), 2000, pp. 743–750.

[92] M.R. Prasad, A. Biere, A. Gupta, A survey of recent advances in SAT-based formal verification, Int. J. Softw. Tools Technol. Transf. 7 (2) (2005) 156–173.

[93] J. Quinonero-Candela, C.E. Rasmussen, C.K. Williams, Approximation methods for Gaussian process regression, in: Léon Bottou, Olivier Chapelle, Dennis DeCoste, Jason Weston (Eds.), Large-Scale Kernel Machines, MIT Press, 2007, pp. 203–223.

[94] C.E. Rasmussen, C.K.I. Williams, Gaussian Processes for Machine Learning, MIT Press, 2006.

[95] J.R. Rice, The algorithm selection problem, Adv. Comput. 15 (1976) 65–118.

[96] E. Ridge, D. Kudenko, Tuning the performance of the MMAS heuristic, in: Proceedings of the International Workshop on Engineering Stochastic Local Search Algorithms (SLS'2007), in: LNCS, vol. 4638, Springer-Verlag, 2007, pp. 46–60.

[97] M. Roberts, A. Howe, Learned models of performance for many planners, in: ICAPS 2007 Workshop AI Planning and Learning, 2007.

[98] J. Sacks, W.J. Welch, T.J. Welch, H.P. Wynn, Design and analysis of computer experiments, Stat. Sci. 4 (4) (1989) 409–423.

[99] T.J. Santner, B.J. Williams, W.I. Notz, The Design and Analysis of Computer Experiments, Springer, 2003.

[100] J. Schmee, G.J. Hahn, A simple method for regression analysis with censored data, Technometrics 21 (4) (1979) 417–432.

[101] M. Schmidt, minfunc, http://www.di.ens.fr/~mschmidt/Software/minFunc.html, 2012. Last accessed on August 5, 2012.

[102] M.R. Segal, Regression trees for censored data, Biometrics 44 (1) (1988) 35–47.

[103] J. Sherman, W.J. Morrison, Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix (abstract), Ann. Math. Stat. 20 (1949) 621.

[104] K. Smith-Miles, Cross-disciplinary perspectives on meta-learning for algorithm selection, ACM Comput. Surv. 41 (1) (2009) 6:1–6:25.

[105] K. Smith-Miles, L. Lopes, Measuring instance difficulty for combinatorial optimization problems, Comput. Oper. Res. 39 (5) (2012) 875–889.

[106] K. Smith-Miles, T. Tan, Measuring algorithm footprints in instance space, in: Proceedings of the 2012 Congress on Evolutionary Computation (CEC'12), 2012, pp. 3446–3453.

[107] K. Smith-Miles, J. van Hemert, Discovering the suitability of optimisation algorithms by learning from evolved instances, Ann. Math. Artif. Intell. 61 (2011) 87–104.

[108] K. Smith-Miles, J. van Hemert, X.Y. Lim, Understanding TSP difficulty by learning from evolved instances, in: Proceedings of the 4th Workshop on Learning and Intelligent Optimization (LION'10), in: LNCS, vol. 6073, Springer-Verlag, 2010, pp. 266–280.

[109] M. Soos, CryptoMiniSat 2.5.0. Solver description, SAT Race 2010.

[110] V. Tresp, A Bayesian committee machine, Neural Comput. 12 (11) (2000) 2719–2741.

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

[112] W. Wei, C.M. Li, Switching between two adaptive noise mechanisms in local search for SAT. Solver description, SAT competition, 2009.

[113] E. Weinberger, Correlated and uncorrelated fitness landscapes and how to tell the difference, Biol. Cybern. 63 (1990) 325–336.

[114] N.A. Weiss, A Course in Probability, Addison–Wesley, 2005.

[115] L. Xu, H.H. Hoos, K. Leyton-Brown, Hierarchical hardness models for SAT, in: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07), in: LNCS, vol. 4741, Springer-Verlag, 2007, pp. 696–711.

[116] L. Xu, H.H. Hoos, K. Leyton-Brown, Hydra: automatically configuring algorithms for portfolio-based selection, in: Proceedings of the 25th National Conference on Artificial Intelligence (AAAI'10), 2010, pp. 210–216.

[117] L. Xu, F. Hutter, H. Hoos, K. Leyton-Brown, SATzilla-07: the design and analysis of an algorithm portfolio for SAT, in: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07), in: LNCS, vol. 4741, Springer-Verlag, 2007, pp. 712–727.

[118] L. Xu, F. Hutter, H. Hoos, K. Leyton-Brown, SATzilla2009: an automatic algorithm portfolio for sat. Solver description, SAT competition, 2009.

[119] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT, J. Artif. Intell. Res. 32 (2008) 565–606.

[120] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, Evaluating component solver contributions in portfolio-based algorithm selectors, in: Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12), in: LNCS, vol. 7317, Springer-Verlag, 2012, pp. 228–241.

[121] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, Satzilla2012: Improved algorithm selection based on cost-sensitive classification models, in: Fifteenth International Conference on Theory and Applications of Satisfiability Testing, SAT Challenge 2012: Solver Descriptions, 2012.

[122] E. Zarpas, Benchmarking SAT solvers for bounded model checking, in: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05), in: LNCS, vol. 3569, Springer-Verlag, 2005, pp. 340–354.