

---

# An Efficient Approach for Assessing Hyperparameter Importance

---

**Frank Hutter**

University of Freiburg, Freiburg, GERMANY

FH@INFORMATIK.UNI-FREIBURG.DE

**Holger Hoos**

University of British Columbia, Vancouver, CANADA

HOOS@CS.UBC.CA

**Kevin Leyton-Brown**

University of British Columbia, Vancouver, CANADA

KEVINLB@CS.UBC.CA

## Abstract

The performance of many machine learning methods depends critically on hyperparameter settings. Sophisticated Bayesian optimization methods have recently achieved considerable successes in optimizing these hyperparameters, in several cases surpassing the performance of human experts. However, blind reliance on such methods can leave end users without insight into the relative importance of different hyperparameters and their interactions. This paper describes efficient methods that can be used to gain such insight, leveraging random forest models fit on the data already gathered by Bayesian optimization. We first introduce a novel, linear-time algorithm for computing marginals of random forest predictions and then show how to leverage these predictions within a functional ANOVA framework, to quantify the importance of both single hyperparameters and of interactions between hyperparameters. We conducted experiments with prominent machine learning frameworks and state-of-the-art solvers for combinatorial problems. We show that our methods provide insight into the relationship between hyperparameter settings and performance, and demonstrate that—even in very high-dimensional cases—most performance variation is attributable to just a few hyperparameters.

## 1. Introduction

Machine learning algorithms often behave very differently with different instantiations of their hyperparameters. This is true especially for complex model families, such as deep belief networks (Hinton *et al.*, 2006), convolutional

networks (LeCun *et al.*, 2001), stacked denoising autoencoders (Vincent *et al.*, 2010), and computer vision architectures (Bergstra *et al.*, 2013), all of which have tens up to hundreds of hyperparameters. Since hyperparameter settings often make the difference between mediocre and state-of-the-art performance, and since naive hyperparameter optimization methods, such as grid search, do not scale to many dimensions, there has been a recent surge of interest in more sophisticated hyperparameter optimization methods (Hutter *et al.*, 2011; Bergstra and Bengio, 2012; Bergstra *et al.*, 2011; Snoek *et al.*, 2012; Bardenet *et al.*, 2013). In low-dimensional problems with numerical hyperparameters, the best available hyperparameter optimization methods use Bayesian optimization (Brochu *et al.*, 2009) based on Gaussian process models, whereas in high-dimensional and discrete spaces, tree-based models (Bergstra *et al.*, 2011), and in particular random forests (Hutter *et al.*, 2011; Thornton *et al.*, 2013; Gramacy *et al.*, 2013), are more successful (Eggenberger *et al.*, 2013).

Such modern hyperparameter optimization methods have achieved considerable recent success. For example, Bayesian optimization found a better instantiation of nine convolutional network hyperparameters than a domain expert, thereby achieving the lowest error reported on the CIFAR-10 benchmark at the time (Snoek *et al.*, 2012). In high-dimensional hyperparameter optimization problems, recent success stories include (1) a new best result for the MNIST rotated background images dataset in 2011 using an automatically configured deep belief network with 32 hyperparameters (Bergstra *et al.*, 2011); (2) a complex vision architecture with 238 hyperparameters that can be instantiated to yield state-of-the-art performance for such disparate tasks as face-matching verification, face identification, and object recognition (Bergstra *et al.*, 2013); and (3) AutoWEKA, a framework enabling per-dataset optimization over a 768-dimensional space including all model classes and hyperparameters defined in WEKA (Thornton *et al.*, 2013).

A similar development can be observed in the area of combinatorial optimization, where automated hyperparameter optimization approaches have recently led to substantial improvements of high-performance heuristic algorithms for a wide range of problems, including propositional satisfiability (Hutter *et al.*, 2007; KhudaBukhsh *et al.*, 2009; Tompkins *et al.*, 2011), mixed integer programming (Hutter *et al.*, 2010), AI planning (Vallati *et al.*, 2013), answer set programming (Silverthorn *et al.*, 2012) and the traveling salesman problem (Styles and Hoos, 2013).<sup>1</sup> While traditional hyperparameter optimization methods in that community are based on heuristic search (Adenso-Diaz and Laguna, 2006; Nannen and Eiben, 2007; Hutter *et al.*, 2009; Ansotegui *et al.*, 2009) and racing algorithms (Maron and Moore, 1994; Birattari *et al.*, 2010), recently, Bayesian optimization methods based on random forest models have been shown to compare favourably (Hutter *et al.*, 2011).

The considerable success of Bayesian optimization for determining good hyperparameter settings in machine learning and combinatorial optimization has not yet been accompanied by much work on methods for providing scientists with answers to questions like the following: How important is each of the hyperparameters, and how do their values affect performance? Which hyperparameter interactions matter? How do the answers to these questions depend on the data set under consideration?

The answer to such questions is the key to scientific discoveries, and consequently, recent Bayesian optimization workshops at NIPS have identified these topics as a core area in need of increased attention. Recent work on Bayesian optimization has targeted the case where most hyperparameters are truly unimportant (Chen *et al.*, 2012; Wang *et al.*, 2013), and several applications have yielded evidence that some hyperparameters indeed tend to be much more important than others (Bergstra and Bengio, 2012; Hutter *et al.*, 2013b). However, not much work has been done on quantifying the relative importance of the hyperparameters that *do* matter.

In this paper, we investigate the classic technique of functional analysis of variance (functional ANOVA) (Sobol, 1993; Huang, 1998; Jones *et al.*, 1998; Hooker, 2007) to decompose the variance  $\mathbb{V}$  of a blackbox function  $f : \Theta_1 \times \dots \times \Theta_n \rightarrow \mathbb{R}$  into additive components  $\mathbb{V}_U$  associated with each subset of hyperparameters  $U \subseteq \{1, \dots, n\}$ . In our case,  $f$  is our algorithm’s performance with hyperparameter settings  $\theta$ . As is standard, we learn a predictive model  $\hat{f}$  of  $f$  and partition the variance of  $\hat{f}$ . In order to do this tractably, we must be able to efficiently compute marginalizations of  $\hat{f}$  over arbitrary input dimensions  $T \subseteq \{1, \dots, n\}$ . This has been shown to be possible for

<sup>1</sup>While that community uses the term ‘parameters’ for design choices that need to be instantiated before running an algorithm, here we stick with machine learning nomenclature and use the term ‘hyperparameters’ for these choices throughout.

Gaussian process models  $\hat{f}$  with certain kernels (see, e.g., Jones *et al.*, 1998). However, here, we are most interested in random forest models, since these have been shown to achieve the best performance for model-based optimization in complex hyperparameter spaces, particularly in cases involving categorical and conditional hyperparameters (Thorn-ton *et al.*, 2013; Eggen-sperger *et al.*, 2013). To date, efficient marginalizations had not been available for random forest models, forcing researchers to revert to sampling-based techniques to compute approximate functional ANOVA decompositions (Gramacy *et al.*, 2013). Here, we provide the first efficient and exact method for deriving functional ANOVA sensitivity indices for random forests.

When applying this new method to quantify the importance of the hyperparameters of machine learning algorithms and combinatorial optimization procedures, following Hutter *et al.* (2011), we consider a setting slightly more general than blackbox function optimization: given an algorithm  $A$  with configuration space  $\Theta$ , a set of training scenarios  $\pi_1, \dots, \pi_k$ , and a performance metric  $m(\theta, \pi)$  capturing  $A$ ’s performance with hyperparameter configurations  $\theta \in \Theta$  on scenario  $\pi$ , find a configuration  $\theta \in \Theta$  that minimizes  $m$  over  $\pi_1, \dots, \pi_k$ , *i.e.*, that minimizes the function

$$f(\theta) := \sum_{i=1}^k m(\theta, \pi_i).$$

In the case of hyperparameter optimization of machine learning algorithms, the  $\pi_i$  are typically cross-validation folds, and for combinatorial problem solving procedures, they are problem instances deemed representative for the kind of instances we aim to optimize performance for.

In the following, we first introduce our new, linear-time algorithm for computing the marginals of random forest predictions (Section 2). We then show how this algorithm can be leveraged to tractably identify main and (low-order) interaction effects within the functional ANOVA framework (Section 3). Finally, we demonstrate the power of this approach through an extensive experimental evaluation, using highly parametric machine learning frameworks and combinatorial solvers for NP-hard problems (Section 4).

## 2. Efficient Marginal Performance Predictions

Algorithm designers wanting to manually assess hyperparameter importance often investigate the local neighbourhood of a given hyperparameter configuration: vary one hyperparameter at a time and measure how performance changes. Note that the only information obtained with this analysis is how different hyperparameter values perform *in the context of a single instantiation* of the other hyperparameters. Optimally, algorithm designers would like to know how their hyperparameters affect performance in general, not just in the context of a single fixed instantiation of the

remaining hyperparameters, but across all their instantiations. Unfortunately, performing algorithm runs for all these instantiations is infeasible in all but the easiest cases, since there are  $D^k$  such instantiations of  $k$  discrete hyperparameters with domain size  $D$ . (Continuous hyperparameters are even worse, having infinitely many instantiations.) As it turns out, an approximate analysis based on predictive models can be used to solve this problem and quantify the performance of a hyperparameter instantiation in the context of *all instantiations of the other hyperparameters*. In this section, we will show that this *marginal performance* of a partial hyperparameter instantiation can be predicted by computing the required exponential (or infinite) sum of predictions in linear time. We first cover some notation and define the problem formally. Then, we introduce an algorithm to predict this marginal performance using random forests and prove its correctness and linear time complexity.

## 2.1. Problem Definition

We begin with some basic definitions. Let  $A$  be an algorithm having  $n$  hyperparameters with domains  $\Theta_1, \dots, \Theta_n$ . We use integers to denote the hyperparameters, and  $N$  to refer to the set  $\{1, \dots, n\}$  of all hyperparameters of  $A$ .

**Definition 1** (Configuration space  $\Theta$ ).  $A$ 's configuration space is  $\Theta = \Theta_1 \times \dots \times \Theta_n$ .

**Definition 2** (Hyperparameter Instantiation). A complete instantiation of an algorithm's  $n$  hyperparameters is a vector  $\theta = \langle \theta_1, \dots, \theta_n \rangle$  with  $\theta_i \in \Theta_i$ . We also refer to complete hyperparameter instantiations as hyperparameter configurations. A partial instantiation of a subset  $U = \{u_1, \dots, u_m\} \subseteq N$  of  $A$ 's hyperparameters is a vector  $\theta_U = \langle \theta_{u_1}, \dots, \theta_{u_m} \rangle$  with  $\theta_{u_i} \in \Theta_{u_i}$ .

The extension set of a partial hyperparameter instantiation is the set of hyperparameter configurations that are consistent with it.

**Definition 3** (Extension Set). Let  $\theta_U = \langle \theta_{u_1}, \dots, \theta_{u_m} \rangle$  be a partial instantiation of the hyperparameters  $U = \{u_1, \dots, u_m\} \subseteq N$ . The extension set  $X(\theta_U)$  of  $\theta_U$  is then the set of hyperparameter configurations  $\theta_{N|U} = \langle \theta'_1, \dots, \theta'_n \rangle$  such that  $\forall j (j = u_k \Rightarrow \theta'_j = \theta_{u_k})$ .

To handle sets of hyperparameter configurations with a mix of continuous and categorical hyperparameters, we define the range size of a set.

**Definition 4** (Range size). The range size  $\|S\|$  of an empty set  $S$  is defined as 1; for other finite  $S$ , the range size equals the cardinality:  $\|S\| = |S|$ . For closed intervals  $S = [l, u] \subset \mathbb{R}$  with  $l < u$ ,  $\|S\| = u - l$ . For cross-products  $S = S_1 \times \dots \times S_k$ ,  $\|S\| = \prod_{i=1}^k \|S_i\|$ .

The probability density of a uniform distribution over  $X(\theta_U)$  is then simply  $1/\|X(\theta_U)\| = 1/\|\Theta_T\|$ , where  $T = \{t_1, \dots, t_k\} = N \setminus U$  and  $\Theta_T = \Theta_{t_1} \times \dots \times \Theta_{t_k}$ .

Now, we can define the marginal (predicted) performance of a partial instantiation  $\theta_U$  as the expected (predicted) performance of  $A$  across  $X(\theta_U)$ .

**Definition 5** (Marginal performance). Let  $A$ 's (true) performance be  $y : \Theta \mapsto \mathbb{R}$ ,  $U \subseteq N$ , and  $T = N \setminus U$ .  $A$ 's marginal performance  $a_U(\theta_U)$  is then defined as

$$\begin{aligned} a_U(\theta_U) &= \mathbb{E}[y(\theta_{N|U}) \mid \theta_{N|U} \in X(\theta_U)] \\ &= \frac{1}{\|\Theta_T\|} \int y(\theta_{N|U}) d\theta_T. \end{aligned}$$

Similarly,  $A$ 's marginal predicted performance  $\hat{a}_U(\theta_U)$  under a model  $\hat{y} : \Theta \rightarrow \mathbb{R}$  is

$$\hat{a}_U(\theta_U) = \frac{1}{\|\Theta_T\|} \int \hat{y}(\theta_{N|U}) d\theta_T. \quad (1)$$

Note that if the predictive model  $\hat{y}$  has low error on average across the configuration space, the difference between predicted and true marginal performance will also be low.

## 2.2. Efficient Computation of Marginal Predictions in Random Forests

In this section, we show that when using random forest predictors  $\hat{y}$ , the marginal predicted performance  $\hat{a}_U(\theta_U)$  defined in Eq. 1 can be computed exactly in linear time. The fact that this can be done for random forests is important for our application setting, since random forests yield the best performance predictions for a broad range of highly parameterized algorithms (Hutter *et al.*, 2014).

Random forests (Breiman, 2001) are ensembles of regression trees. Each regression tree partitions the input space through sequences of branching decisions that lead to each of its leaves. We denote this partitioning as  $\mathcal{P}$ . Each equivalence class  $P_i \in \mathcal{P}$  is associated with a leaf of the regression tree and with a constant  $c_i$ . Let  $\Theta_j^{(i)} \subset \Theta_j$  denote the subset of domain values of hyperparameter  $j$  that is consistent with the branching decisions leading to the leaf associated with  $P_i$ . Then, for trees with axis-aligned splits,  $P_i$  is simply the Cartesian product

$$P_i = \Theta_1^{(i)} \times \dots \times \Theta_n^{(i)}. \quad (2)$$

The predictor  $\hat{y} : \Theta \rightarrow \mathbb{R}$  encoded by the regression tree is

$$\hat{y}(\theta) = \sum_{P_i \in \mathcal{P}} \mathbb{I}(\theta \in P_i) \cdot c_i, \quad (3)$$

where  $\mathbb{I}(x)$  is the indicator function. Random forests simply predict the average of their individual regression trees.

Our approach for computing marginal predictions  $\hat{a}_U(\theta_U)$  of a random forest works in two phases: a preprocessing phase that has to be carried out only once and a prediction phase that has to be carried out once per requested marginal

**Algorithm 1: ComputePartitioning**( $\Theta, \mathcal{T}, i, \Theta^{(i)}$ )

---

**Input** :  $\Theta = \Theta_1 \times \dots \times \Theta_n$ , a configuration space;  $\mathcal{T}$ , a regression tree partitioning  $\Theta$ ;  $i$ , a node;  
 $\Theta^{(i)} = \Theta_1^{(i)} \times \dots \times \Theta_n^{(i)}$ ,  $i$ 's partition of  $\Theta$

**Output** : Partitioning  $\mathcal{P} = \{P_1, \dots, P_k\}$  of  $\Theta^{(i)}$

- 1 **if** node  $i$  is a leaf **then return**  $\{\Theta^{(i)}\}$
- 2 **else**
- 3     Let  $v$  be the hyperparameter that node  $i$  splits on
- 4     Follow the splitting rule defined by node  $i$  to partition  $\Theta_v^{(i)}$  into newly created sets  $\Theta_v^{(l)}$  and  $\Theta_v^{(r)}$  for its left and right child  $l$  and  $r$ , respectively
- 5      $\mathcal{P}_l \leftarrow \text{ComputePartitioning}(\Theta, \mathcal{T}, l,$   
 $\Theta_1^{(i)} \times \dots \times \Theta_{v-1}^{(i)} \times \Theta_v^{(l)} \times \Theta_{v+1}^{(i)} \times \dots \times \Theta_n^{(i)})$
- 6      $\mathcal{P}_r \leftarrow \text{ComputePartitioning}(\Theta, \mathcal{T}, r,$   
 $\Theta_1^{(i)} \times \dots \times \Theta_{v-1}^{(i)} \times \Theta_v^{(r)} \times \Theta_{v+1}^{(i)} \times \dots \times \Theta_n^{(i)})$
- 7     **return**  $\mathcal{P}_l \cup \mathcal{P}_r$

---

prediction. Both phases require only linear time given a random forest model as input (constructing the random forest is a separate problem, but is also cheap: for  $T$  data points of dimensionality  $n$ , it is  $O(n \cdot T^2 \log T)$  in the worst case and  $O(n \cdot T \log^2 T)$  in the more realistic best case of balanced trees (Hutter *et al.*, 2014)).

The key idea behind our algorithm is to exploit the fact that each of the regression trees in a given forest defines a partitioning  $\mathcal{P}$  of the configuration space  $\Theta$ , with piecewise constant predictions  $c_i$  in each  $P_i \in \mathcal{P}$ , and that the problem of computing sums over an arbitrary number of configurations thus reduces to the problem of identifying the ratio of configurations that fall into each partition.

We first show that, given a partitioning  $\mathcal{P}$ , we can compute marginal predictions as a linear sum over entries in the leaves.

**Theorem 6.** *Given the partitioning  $\mathcal{P}$  of a regression tree  $\mathcal{T}$  that defines a predictor  $\hat{y} : \Theta \mapsto \mathbb{R}$ , and a partial instantiation  $\theta_U$  of  $\Theta$ 's hyperparameters  $N$ ,  $\mathcal{T}$ 's marginal prediction  $\hat{a}_U(\theta_U)$  can be computed as*

$$\hat{a}_U(\theta_U) = \sum_{P_i \in \mathcal{P}} \frac{\|\Theta_{N \setminus U}^{(i)}\|}{\|\Theta_{N \setminus U}\|} \mathbb{I}(\theta_U \in \Theta_U^{(i)}) \cdot c_i.$$

All proofs are provided in Appendix A. Using Theorem 6, we can compute arbitrary marginals by a simple iteration over the partitions by counting the ratio of hyperparameter configurations falling into each partition. However, regression trees do not normally provide explicit access to these partitions; they are defined implicitly through the tree structure. Since we need to represent the partitioning explicitly, one may worry about space complexity. Indeed, if we stored the values  $\Theta_j^{(i)}$  for each leaf  $i$  and categorical hyperparameter  $j$  (as well as lower and upper bounds for continuous hyperparameters), for a random forest with  $B$  trees of  $L$  leaves each, and  $n$  categorical hyperparameters with domain

size at most  $D$ , we would end up with space complexity of  $\Theta(B \cdot L \cdot n \cdot D)$ . In the largest of the practical scenarios we consider later in this work (random forests with  $B = 10$  trees of up to  $L = 100\,000$  leaves, configuration spaces with up to  $n = 768$  hyperparameters and domain sizes up to  $D = 20$ ) this would have been infeasible. Instead, we show that Algorithm 1 can compute the partitioning using a pointer-based data structure, reducing the space complexity to  $O(B \cdot L \cdot (D + n))$ . (Alternatively, when space is not a concern, the partitioning can be represented using a bit mask, replacing  $O(\log(D))$  member queries with  $O(1)$  operations and thus reducing the complexity of marginal predictions for single trees from  $O(L \cdot n \log D)$  to  $O(L \cdot n)$ .)

**Theorem 7.** *Given a regression tree  $\mathcal{T}$  with  $L$  leaves and a configuration space  $\Theta$  with  $n$  hyperparameters and categorical domain size at most  $D$ , Algorithm 1 computes  $\mathcal{T}$ 's partitioning of  $\Theta$  in time and space  $O(L \cdot D + L \cdot n)$ .*

To compute marginal predictions of random forests, we average over the marginal predictions of their individual trees. We use the variance across these individual tree predictions to express our model uncertainty.

**Corollary 8.** *Given a random forest with  $B$  trees of up to  $L$  leaves that defines a predictor  $\hat{y} : \Theta \rightarrow \mathbb{R}$  for a configuration space  $\Theta$  with  $n$  hyperparameters and categorical domain size at most  $D$ , the time and space complexity of computing a single marginal of  $\hat{y}$  is  $O(B \cdot L \cdot \max\{D + n, n \log D\})$ . Each additional marginal can be computed in additional space  $O(1)$  and time  $O(B \cdot L \cdot n \log D)$ .*

### 3. Efficient Decomposition of Variance

In this section, we review functional analysis of variance and demonstrate how we can use our efficient marginal predictions with this technique to quantify the importance of an algorithm's individual hyperparameters and of low-order interactions between hyperparameters.

#### 3.1. Functional Analysis of Variance (ANOVA)

Functional analysis of variance (functional ANOVA) is a prominent data analysis method from the statistics literature (Sobol, 1993; Huang, 1998; Jones *et al.*, 1998; Hooker, 2007). While this method has not received much attention in the machine learning community so far, we believe that it offers great value. In a nutshell, ANOVA partitions the observed variation of a response value (here: algorithm performance) into components due to each of its inputs (here: hyperparameters). *Functional ANOVA* decomposes a function  $\hat{y} : \Theta_1 \times \dots \times \Theta_n \rightarrow \mathbb{R}$  into additive components that only depend on subsets of its inputs  $N$ :

$$\hat{y}(\theta) = \sum_{U \subseteq N} \hat{f}_U(\theta_U).$$

**Algorithm 2: QuantifyImportance( $\Theta, \mathcal{T}, K$ )**


---

**Input** :  $\Theta$ , a configuration space with hyperparameters  $N$ ;  
 $\mathcal{T}$ , a regression tree;  $K$ , the maximal order of  
 interaction effects to be computed

**Output** :  $\{\mathbb{F}_U \mid U \subseteq N, |U| \leq K\}$ , the fractions of  
 variance contributed by all hyperparameter subsets  
 up to size  $K$

- 1  $\mathcal{P} \leftarrow \text{ComputePartitioning}(\Theta, \mathcal{T}, 1, \Theta)$  // 1 is  $\mathcal{T}$ 's root
- 2  $\hat{f}_\emptyset \leftarrow \sum_{P_i \in \mathcal{P}} \left( \prod_{j \in N} \|\Theta_j^{(i)}\| / \|\Theta_j\| \right) \cdot c_i$ .
- 3  $\mathbb{V} \leftarrow \sum_{P_i \in \mathcal{P}} \left( \prod_{j \in N} \|\Theta_j^{(i)}\| / \|\Theta_j\| \right) \cdot (c_i - \hat{f}_\emptyset)^2$ .
- 4 **for**  $k = 1, \dots, K$  **do**
- 5     **for all**  $U \in \{U' \subset N, |U'| = k\}$  **do**
- 6          $\mathbb{V}_U \leftarrow 0$
- 7         **for all**  $\theta_U \in \Theta_U$  **do**
- 8              $\hat{a}_U(\theta_U) \leftarrow \sum_{P_i \in \mathcal{P}} \frac{\|\Theta_{N \setminus U}^{(i)}\|}{\|\Theta_{N \setminus U}\|} \mathbb{I}(\theta_U \in$   
 $\Theta_U^{(i)}) \cdot c_i$
- 9              $\hat{f}_U(\theta_U) \leftarrow \hat{a}_U(\theta_U) - \sum_{W \subsetneq U} \hat{f}_W(\theta_W)$
- 10              $\mathbb{V}_U \leftarrow \mathbb{V}_U + 1 / \|\Theta_U\| \cdot \hat{f}_U(\theta_U)^2$
- 11              $\mathbb{F}_U \leftarrow \mathbb{V}_U / \mathbb{V}$
- 12 **return**  $\{\mathbb{F}_U \mid U \subseteq N, |U| \leq K\}$

---

The components  $\hat{f}_U(\theta_U)$  are defined as follows:

$$\hat{f}_U(\theta_U) = \begin{cases} \frac{1}{\|\Theta\|} \int \hat{y}(\theta) d\theta & \text{if } U = \emptyset. \\ \hat{a}_U(\theta_U) - \sum_{W \subsetneq U} \hat{f}_W(\theta_W) & \text{otherwise.} \end{cases} \quad (4)$$

The constant  $\hat{f}_\emptyset$  is the function's mean across its domain. The unary functions  $\hat{f}_{\{j\}}(\theta_{\{j\}})$  are called *main effects* and capture the effect of varying hyperparameter  $j$ , averaging across all instantiations of all other hyperparameters. The functions  $\hat{f}_U(\theta_U)$  for  $|U| > 1$  capture exactly the interaction effects between all variables in  $U$  (excluding all lower-order main and interaction effects of  $W \subsetneq U$ ).

By definition, the variance of  $\hat{y}$  across its domain  $\Theta$  is

$$\mathbb{V} = \frac{1}{\|\Theta\|} \int (\hat{y}(\theta) - \hat{f}_\emptyset)^2 d\theta, \quad (5)$$

and functional ANOVA decomposes this variance into contributions by all subsets of variables (see, e.g., [Hooker, 2007](#), for a derivation):

$$\mathbb{V} = \sum_{U \subsetneq N} \mathbb{V}_U, \quad \text{where } \mathbb{V}_U = \frac{1}{\|\Theta_U\|} \int \hat{f}_U(\theta_U)^2 d\theta_U. \quad (6)$$

The importance of each main and interaction effect  $\hat{f}_U$  can thus be quantified by the fraction of variance it explains:  $\mathbb{F}_U = \mathbb{V}_U / \mathbb{V}$ .

### 3.2. Variance Decomposition in Random Forests

In Algorithm 2, we use our efficient marginal predictions from Section 2.2 to quantify the importance of main and

interaction effects of random forest predictors in the functional ANOVA framework of Eq. 6.

**Theorem 9.** *Given a configuration space  $\Theta$  consisting of  $n$  categorical hyperparameters<sup>2</sup> of maximal domain size  $D$  and a regression tree  $\mathcal{T}$  with  $L$  leaves that defines a predictor  $\hat{y} : \Theta \rightarrow \mathbb{R}$ , Algorithm 2 exactly computes the fractions of variance explained by all subsets  $U$  of  $\Theta$ 's hyperparameters  $N$  of arity up to  $K$ , with space complexity  $O(L \cdot D + L \cdot n)$  and time complexity  $O\left(L \cdot D + \sum_{k=1}^K \binom{n}{k} \cdot D^k (L \cdot n \log d + 2^k)\right)$ .*

To compute hyperparameter importance in random forests, we simply apply Algorithm 2 for each tree and compute means and standard deviations across the trees.

### 3.3. Practical Use for Identifying Hyperparameter Importance Given Logged Performance Data

Note that Theorem 9 assumes regression tree predictors  $\hat{y} : \Theta \rightarrow \mathbb{R}$  that predict the performance of hyperparameter configurations and do not mention training scenarios (such as cross-validation folds in machine learning, or problem instances when optimizing combinatorial solvers). However, the performance data logged in modern Bayesian optimization methods consists of algorithm runs on single training scenarios: ‘‘algorithm run  $t$  used configuration  $\theta_t$  on fold/instance  $\pi_t$  and achieved performance  $y_t$ ’’. We thus construct random forest predictors  $\hat{y}' : \Theta \times \Pi \rightarrow \mathbb{R}$  and use them to predict, for every unique  $\theta$  in our training data, the average performance  $\hat{m}_\theta = 1/m \sum_{i=1}^k m(\theta, \pi_i)$  across training scenarios  $\pi_1, \dots, \pi_k$ . Then, we learn random forests  $\hat{y} : \Theta \rightarrow \mathbb{R}$  using tuples  $(\theta, \hat{m}_\theta)$  as training data. Our variance decomposition then applies directly to the trees in these forests.

While some algorithm designers are interested in the global effect of a hyperparameter (subset) in the context of all other possible hyperparameter settings, others care more about effects in ‘‘good’’ regions of the configuration space, where ‘‘good’’ may have different meanings, such as ‘‘at least as good as the (manually chosen) default’’ or ‘‘in the top  $X\%$  of configurations’’. We can easily support such alternative measures of importance by capping the performance values of each training configuration at a given maximum  $y_{max}$  and learning our random forest on tuples  $(\theta, \min(m_\theta, y_{max}))$  instead of  $(\theta, m_\theta)$ . The consequence is that no configuration is predicted to be worse than  $y_{max}$ , and thus all remaining performance variation characterizes regions of the space with performance better than  $y_{max}$ .

<sup>2</sup>For continuous hyperparameters  $j$  with  $\Theta_j = [l_j, u_j]$ , we have to sum over all intervals of  $[l_j, u_j]$  defined by the split points in  $\bigcup_{P_i \in \mathcal{P}} \{\min \Theta_j^{(i)}, \max \Theta_j^{(i)}\}$ . The number of such intervals can in principle grow as large as the number of leaves, leading to worst-case time complexity  $O\left(L \cdot D + \sum_{k=1}^K \binom{n}{k} \cdot L^k (L \cdot n \log d + 2^k)\right)$ .

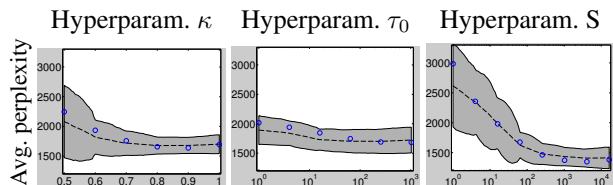


Figure 1. Main effects for Online LDA’s perplexity, one plot per hyperparameter (identified by the column header). Each plot shows marginal perplexity achieved when varying the hyperparameter’s value across the x-axis. The dashed black line and grey-shaded area indicate predicted marginals (mean  $\pm$  one standard deviation). The blue circles denote true marginals.

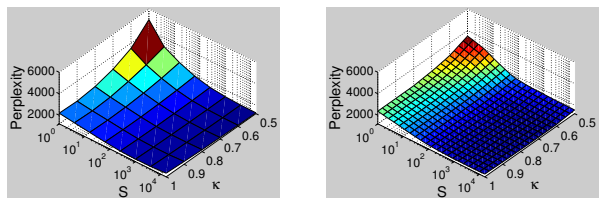
## 4. Empirical Evaluation

We demonstrate our techniques by assessing the importance of hyperparameters of various machine learning algorithms and solvers for NP-hard problems. Additional details and results for all experiments are provided in Appendix B.

### 4.1. Evaluation on Ground-Truth Data

Our first experiment utilizes data gathered with an online variational Bayes algorithm for Latent Dirichlet Allocation (Online LDA) (Hoffman *et al.*, 2010) and made available as part of a previous study in Bayesian optimization for the hyperparameters of this algorithm (Snoek *et al.*, 2012). Complete ground truth data is available for a grid of three hyperparameters ( $\kappa$ ,  $\tau_0$ , and  $S$ ), discretized to 6, 6, and 8 values, respectively;  $\kappa$  controls how quickly information is forgotten,  $\tau_0 > 0$  downweights early iterations, and  $S$  concerns the size of mini-batches. For each of the  $6 \times 8 \times 6 = 288$  grid points, we know the algorithm’s performance score (perplexity) and its training time.

Let us assume we only have available a subset of 100 data points randomly sampled out of the 288. We can then fit a model (here, a random forest) and marginalize out the effects of all hyperparameters but one in the model. Using our efficient marginalization techniques from Section 2, this approach also scales to high dimensions. Figure 1 shows the marginal perplexity that is achieved by setting each hyperparameter to a certain value (and averaging over all instantiations of the others). Clearly, the batch size hyperparameter  $S$  is marginally most important, with large batch sizes consistently yielding lower perplexity. Indeed, functional ANOVA reveals that the batch size hyperparameter by itself is responsible for 65% of the variability of perplexity across the entire space. Another 18% are due to an interaction effect between  $S$  and  $\kappa$ , which is visualized in Figure 2. From this figure, we note that the  $\kappa$  hyperparameter is much more important for small batch sizes than for large ones—an interaction effect that cannot be captured by single marginals. Figures 1 and 2 also compare true marginals (computed from the complete grid data) *vs* our predictions, verifying that the two are closely aligned.



(a) True interaction effect (on the measured grid) (b) Predicted interaction effect (on a finer grid)

Figure 2. Interaction effect for Online LDA’s hyperparameters  $\kappa$  and  $S$  for predicting perplexity.

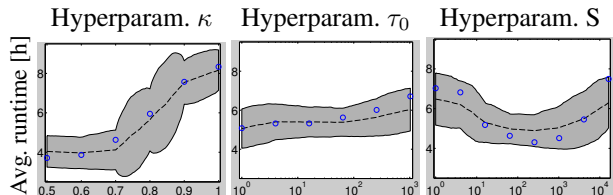


Figure 3. Main effects for Online LDA’s runtime.

Since we also have data on the algorithm’s runtime with different hyperparameter settings, we can carry out exactly the same analysis to assess how runtime depends on hyperparameter settings. As the results in Figure 3 show, hyperparameter  $\kappa$  most influences runtime (causing 54% of the runtime variation), followed by the batch size hyperparameter (causing 21% of the runtime variation). The batch size hyperparameter shows an interesting pattern, with high runtimes for very low or very high values and a sweet spot for intermediate values. Combining this finding with the results from Figure 1 shows that batch sizes around 1 000 yield a good compromise of speed and accuracy.

Overall, in this experiment on ground truth data we observed that our predicted marginals and functional ANOVA analyses are accurate and can give interesting insights even for low-dimensional hyperparameter spaces.

### 4.2. Evaluation on WEKA’s Hyperparameter Space

We now demonstrate how to use our framework for an exploratory analysis in a very high-dimensional hyperparameter space, that of the machine learning framework WEKA (Hall *et al.*, 2009). We use the hyperparameter space defined by the recent Auto-WEKA framework (Thornton *et al.*, 2013), which includes not only numerical hyperparameters of each of the models implemented in WEKA, but also discrete choices about which model to use, meta-classifiers, ensemble classifiers, and a multitude of feature selection mechanisms. With 768 hyperparameters, Auto-WEKA constitutes the largest configuration space of which we are aware, and it is precisely for this reason that we chose to study it: at this dimensionality, it is virtually impossible to manually keep track of each individual hyperparameter.

Auto-WEKA uses tree-based Bayesian optimization meth-

Dataset	# data	time	1st	2nd	3rd
YEAST	12823	187s	class (31%)	feature-s (10%)	baseclass (4%)
AMAZON	179	3s	class (58%)	baseclass (18%)	feature-s (5%)
MNIST BASIC	129	3s	class (55%)	baseclass (15%)	feature-s (6%)
KDD09-APPENTENCY	682	7s	class (41%)	baseclass (16%)	feature-s (3%)
CIFAR-10	99	2s	class (53%)	baseclass (23%)	feature-s (4%)

Table 1. Top 3 most important hyperparameters of Auto-WEKA for each of 5 datasets, based on data from one SMAC run. We also list the number of data points (*i.e.*, ⟨WEKA configuration, performance⟩ pairs) gathered by SMAC, as well as fANOVA’s run-time using this data. We give means computed across 25 repetitions (each based on a different SMAC run).

ods, such as TPE (Bergstra *et al.*, 2011) and SMAC (Hutter *et al.*, 2011), to search WEKA’s joint space of models and hyperparameters. Hyperparameter settings are evaluated by running the respective instantiations of WEKA on one or more cross-validation folds, and the resulting performance values are used to inform a sequential search process. Thornton *et al.* (2013) made available the performance data for each of 21 datasets gathered by SMAC (the method yielding the best Auto-WEKA performance) during its optimization process.

Table 1 lists a representative subset of those 21 data sets, along with the average number of data points gathered in a single SMAC run for the respective data set. With this data as input, we ran our functional ANOVA approach to identify the three most important hyperparameters for each dataset; we list these in Table 1 along with the fraction of variance explained by them. Not surprisingly, the most important hyperparameter was the model class used. Depending on the dataset, the second-most important hyperparameter concerned either feature selection or the base classifier to be used inside a meta classifier.

Figure 4 shows the marginal performance of each model class in more detail for two representative datasets, revealing that different model classes perform very differently on different data sets. Note that performance, as shown in these plots, is marginalized over all possible instantiations of a given model, giving an advantage to methods that are robust with respect to their hyperparameter settings.

These results demonstrate that our framework can be used out of the box for very high-dimensional hyperparameter spaces with mixed continuous/discrete hyperparameters.

### 4.3. Evaluation for Combinatorial Problem Solvers

To demonstrate the versatility of our variance decomposition methods, we also applied them to assess hyperparameter importance of seven highly parametric state-of-the-art solvers for prominent combinatorial problems; in particular, we considered ten benchmarks from propositional satisfiability (SAT), mixed integer programming (MIP) and answer set programming (ASP), covering applications in formal verification, industrial process optimization, computational

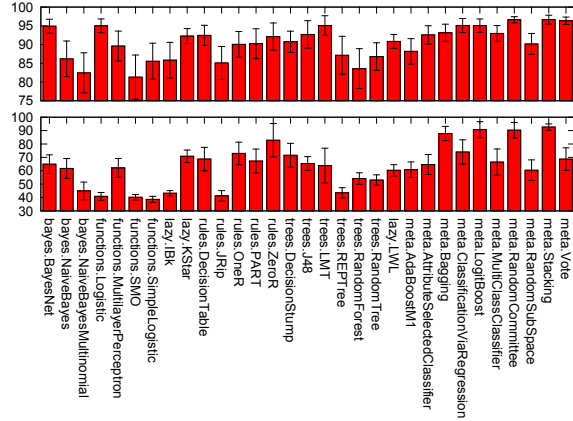


Figure 4. Main effect of Auto-WEKA’s model class hyperparameter for CIFAR-10 (top) and MNIST (bottom).

Scenario	Raw Performance		Impr. over 25% quant.		Impr. over def	
	Main	Pairwise	Main	Pairwise	Main	Pairwise
SPEAR-BMC	88% (2s)	4% (112s)	50% (1s)	15% (36s)	26% (0s)	20% (23s)
SPEAR-SWV	76% (6s)	8% (348s)	19% (1s)	21% (80s)	74% (4s)	11% (250s)
CRYPTOMINISAT-BMC	28% (1s)	18% (62s)	31% (1s)	20% (39s)	6% (0s)	11% (5s)
CRYPTOMINISAT-SWV	37% (4s)	33% (182s)	9% (1s)	19% (44s)	24% (2s)	35% (70s)
SPARROW-3SAT1k	78% (0s)	15% (0s)	53% (0s)	31% (0s)	31% (0s)	34% (0s)
SPARROW-5SAT500	65% (0s)	28% (0s)	57% (0s)	34% (0s)	66% (0s)	27% (0s)
CAPTAINJACK-3SAT1k	42% (9s)	9% (1321s)	21% (4s)	9% (599s)	37% (6s)	9% (832s)
CAPTAINJACK-5SAT500	20% (6s)	11% (917s)	18% (2s)	12% (308s)	26% (5s)	12% (726s)
SATENSTEIN-3SAT1k	45% (6s)	37% (845s)	23% (2s)	27% (296s)	27% (3s)	29% (334s)
SATENSTEIN-5SAT500	33% (9s)	45% (1155s)	16% (3s)	32% (379s)	22% (5s)	49% (648s)
CPLX-RCW	58% (5s)	6% (713s)	16% (1s)	33% (199s)	6% (1s)	15% (127s)
CPLX-CORLAT	31% (29s)	7% (4361s)	16% (10s)	22% (1427s)	30% (22s)	16% (3129s)
CPLX-Regions200	61% (68s)	19% (10416s)	26% (26s)	33% (3476s)	13% (22s)	27% (2787s)
CPLX-CLS	55% (143s)	5% (21502s)	2% (43s)	4% (5725s)	5% (53s)	15% (6047s)
CLASP-WeightedSeq	46% (13s)	13% (2368s)	27% (5s)	20% (858s)	30% (6s)	20% (1047s)
CLASP-Riposte	39% (103s)	8% (18518s)	10% (68s)	3% (12213s)	15% (82s)	3% (14362s)

Table 2. Fractions of variance explained by main effects and pairwise interaction effects, and total time required to compute all of them. Left: effects for explaining raw performance; Middle: effects for explaining improvements over 25% quantile; Right: effects for explaining improvements over the default.

sustainability and database query optimization.

As for hyperparameter optimization, we carried out an experiment with known ground-truth data for optimizing a combinatorial problem solver (SPARROW, on its two benchmarks in Table 2). The results were qualitatively similar to those from Section 4.1 and are reported in Appendix B.3.

Next, we computed all main and interaction effects for all solver/benchmark combinations we considered; the left third of Table 2 summarizes the results. We note that in all cases, the main effects accounted for a substantial fraction of the overall performance variation (20–88%). Since single hyperparameter effects are easier for humans to understand than complex interaction effects, this is an encouraging finding. These main effects were computed within seconds, meaning that algorithm designers could use our approach interactively. Finally, pairwise interaction effects were also important in several cases, explaining up to 45% of the overall performance variation.

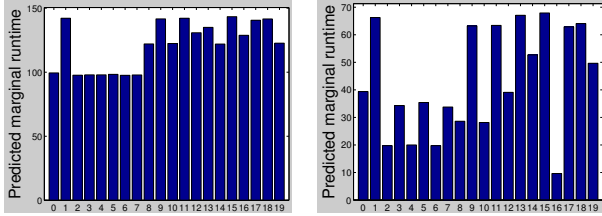


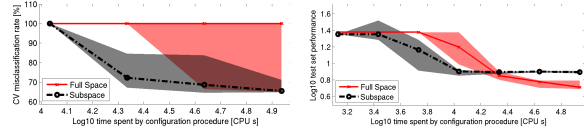
Figure 5. Main effect of SPEAR’s variable selection heuristic (with 20 possible values) for two different instance distributions. Left: BMC; right: SWV.

One particularly interesting case in Table 2 is SPEAR’s performance on bounded model checking (BMC) instances. Here, 87% of the variance was explained by a single hyperparameter, SPEAR’s variable selection heuristic. Figure 5 (left) shows that several standard activity heuristics (labelled 0,2,3,4,5,6,7) performed well for this dataset, whereas other ad-hoc heuristics performed poorly. In contrast, for SPEAR’s performance on software verification (SWV) instances (see Figure 5, right side), one of the heuristics initially suspected to perform poorly turned out to be very effective. Before seeing these results, SPEAR’s developer did not have any intuition about which variable selection heuristic would work well for SWV; in particular, he did not know whether selecting variables in the order they were created (option 16, clearly the best choice) or in reverse order (option 17, one of the worst choices) would be preferable (personal communication). Our result helped him realize that the SAT-encoding used in these SWV instances creates important propositional variables first.

Next, we evaluated hyperparameter importance in the “good” parts of the space, as described in Section 3.3. We used two alternative notions of good configurations: (1) being in the top 25% in terms of performance, and (2) beating the algorithm default. Table 2 (middle and right) shows that single marginals still explain a sizable fraction of the variance in this good part of the space, but less than in the entire space. A closer inspection of extreme cases, such as CPLEX-CLS, showed that in these cases, most of the overall variance was explained by one or more hyperparameters that were best left at their defaults. However, for explaining improvements over the default, these hyperparameters were useless, since none of their values achieved such improvements.

#### 4.4. Configuration in the Subspace of Important Hyperparameters

We can also use our variance decomposition techniques to identify a *hyperparameter subspace* that captures most of the potential for improvements over a good configuration (e.g., the default or the 25% quantile as in Section 4.3). To verify that, by doing so, we indeed capture the potential for improvement, we compare the results found by running the algorithm configuration SMAC in that subspace and in the full space. We performed this experiment for a neural net-



(a) Neural network on CIFAR-10-SMALL (b) CPLEX on CLS

Figure 6. Configuration results with SMAC on full and reduced hyperparameter spaces.

work (8 hyperparameters; subspace: 3 hyperparameters) and for the mixed integer programming solver CPLEX (76 hyperparameters; subspace: 10 hyperparameters). As the results in Figure 6 illustrate, configuration in the subspace yielded good results and often did so faster than in the full space. Note that these results do *not* imply that we have defined an improved configuration procedure for small configuration budgets: the models we used to identify the important subspaces were fit on data gathered by first running SMAC in the complete space. However, they *do* confirm that our functional ANOVA approach rates as important hyperparameters that capture the potential for improvement.

## 5. Conclusion

In this work, we introduced an efficient approach for assessing the importance of the inputs to a blackbox function, and applied it to quantify the effect of algorithm hyperparameters. We first derived a novel linear-time algorithm for computing marginal predictions over arbitrary input dimensions in random forests and then showed how this algorithm can be used to quantify the importance of main effects and interaction effects through a functional ANOVA framework. We empirically validated our approach on performance data from several well-known machine learning frameworks and from state-of-the-art solvers for several prominent combinatorial problems. We confirmed our predictions using ground truth data and showed how our approach can be used to gain insights into the relevance of hyperparameters. We also demonstrated that performance variability is often largely caused by few hyperparameters that define a subspace to which we can restrict configuration.

The methods introduced in this work offer a principled, scientific way for algorithm designers and users to gain deeper insights into the way in which design choices controlled by hyperparameters affect the overall performance of a given algorithm. In future work, we plan to extend our approach to detect dominated hyperparameter values and interactions between instance characteristics and hyperparameter settings. We also plan to develop configuration procedures that determine important hyperparameters on the fly and exploit this information to speed up the optimization process.

Our implementation, along with a quick start guide showing how to apply it to your own algorithms, is publicly available at [www.automl.org/fanova](http://www.automl.org/fanova).



## References

- B. Adenso-Diaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114, Jan–Feb 2006.
- K. Ahmadzadeh, B. Dilkina, C.P. Gomes, and A. Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Proc. of CP-10*, pages 514–521, 2010.
- C. Ansotegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of solvers. In *Proc. of CP-09*, pages 142–157, 2009.
- A. Atamtürk and J. C. Muñoz. A study of the lot-sizing polytope. *Mathematical Programming*, 99:443–465, 2004.
- D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In *Proc. of CAV-07*, pages 366–378, 2007.
- D. Babić and F. Hutter. Spear theorem prover. Solver description, SAT competition, 2007.
- A. Balint, A. Fröhlich, D. A. D. Tompkins, and H. H. Hoos. Sparrow2011. Solver description, SAT competition, 2011.
- R. Bardenet, M. Brendel, B. Kégl, and M. Sebag. Collaborative hyperparameter tuning. In *Proc. of ICML-13*, 2013.
- J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13:281–305, 2012.
- J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. In *Proc. of NIPS-11*, 2011.
- J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proc. of ICML-12*, pages 115–123, 2013.
- M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, editors, *Empirical Methods for the Analysis of Optimization Algorithms*, chapter F-race and iterated F-race: An overview. Springer, 2010.
- L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Technical Report UBC TR-2009-23 and arXiv:1012.2599v1, Dept. of Computer Science, Univ. of British Columbia, 2009.
- B. Chen, R.M. Castro, and A. Krause. Joint optimization and variable selection of high-dimensional Gaussian processes. In *Proc. of ICML-12*, 2012.
- K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS Workshop on Bayesian Optimization in Theory and Practice*, 2013.
- M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proc. of IJCAI-07*, pages 386–392, 2007.
- C.P. Gomes, W. van Hoeve, and A. Sabharwal. Connections in networks: a hybrid approach. In *Proc. of CPAIOR-08*, pages 303–307, 2008.
- R.B. Gramacy, M. Taddy, and S.M. Wild. Variable selection and sensitivity analysis using dynamic trees, with an application to computer code performance tuning. *Ann. Appl. Stat.*, 7(1):51–80, 2013.
- M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.
- G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7), July 2006.
- M. D. Hoffman, D. M. Blei, and F. R. Bach. Online learning for latent Dirichlet allocation. In *Proc. of NIPS-10*, pages 856–864, 2010.
- G. Hooker. Generalized functional ANOVA diagnostics for high dimensional functions of dependent variables. *Journal of Computational and Graphical Statistics*, 16(3), 2007.
- J. Z. Huang. Projection estimation in multiple regression with application to functional anova models. *The Annals of Statistics*, 26(1):242–272, 1998.
- F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proc. of FMCAD-07*, pages 27–34, 2007.
- F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *JAIR*, 36:267–306, October 2009.
- F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR-10*, pages 186–202, 2010.
- F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, pages 507–523, 2011.
- F. Hutter, H. Hoos, and K. Leyton-Brown. Bayesian optimization with censored response data. *CoRR*, abs/1310.1947, 2013.
- F. Hutter, H. H. Hoos, and K. Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *Proc. of LION-7*, pages 364–381, 2013.
- F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206(0):79 – 111, 2014.
- IBM Corp. IBM ILOG CPLEX Optimizer. <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>, 2014. Last accessed on January 6, 2014.
- D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998.
- A. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proc. of IJCAI-09*, pages 517–524, 2009.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In S. Haykin and B. Kosko, editors, *Intelligent Signal Processing*, pages 306–351. IEEE Press, 2001.
- K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proc. of EC-00*, pages 66–76, 2000.

- O. Maron and A. Moore. Hoeffding races: accelerating model selection search for classification and function approximation. In *Proc. of NIPS-94*, pages 59–66, April 1994.
- V. Nannen and A.E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In *Proc. of IJCAI-07*, pages 975–980, 2007.
- B. Silverthorn, Y. Lierler, and M. Schneider. Surviving solver sensitivity: An ASP practitioner’s guide. In *Proc. of ICLP-LIPICS-12*, pages 164–175, 2012.
- J. Snoek, H. Larochelle, and R.P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NIPS-12*, pages 2960–2968, 2012.
- I. M. Sobol. Sensitivity estimates for nonlinear mathematical models. *Mathematical Modeling and Computational Experiment*, 1(4):407–414, 1993.
- M. Soos. CryptoMiniSat 2.5.0. Solver description, SAT Race, 2010.
- J. Styles and H. H. Hoos. Ordered racing protocols for automatically configuring algorithms for scaling performance. In *Proc. of GECCO-13*, pages 551–558, 2013.
- C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. AutoWEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proc. of KDD-13*, pages 847–855, 2013.
- D. A. D. Tompkins, A. Balint, and H. H. Hoos. Captain Jack: new variable selection heuristics in local search for SAT. In *Proc. of SAT-11*, pages 302–316, 2011.
- M. Vallati, C. Fawcett, A. Gerevini, H. H. Hoos, and A. Saetti. Automatic generation of efficient domain-optimized planners from generic parametrized planners. In *Proc. of SOCS-13*, pages 184–192, 2013.
- P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *JMLR*, 11:3371–3408, December 2010.
- Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. de Freitas. Bayesian optimization in high dimensions via random embeddings. In *Proc. of IJCAI-13*, pages 1778–1784, 2013.
- E. Zarpas. Benchmarking SAT solvers for bounded model checking. In *Proc. of SAT-05*, pages 340–354. Springer Verlag, 2005.

## A. Proofs

### A.1. Proof of Theorem 6

We start the algebraic proof below by reiterating the definition of  $\hat{a}_U(\theta_U)$  from (1). To show (7), we plug in the definition of regression trees from (3) for  $\hat{y}(\theta)$ . To show (8), we apply the definition of partition  $P_i$  from (2) and then group terms for  $j \in U$  and  $j \in T$ , respectively. To show (9) we simply rearrange terms. Finally, to show (10), we exploit that  $\Theta_T^{(i)} \subseteq \Theta_T$  to replace the integral  $\int 1/\|\Theta_T\| \cdot \mathbb{I}(\theta_T \in \Theta_T^{(i)}) d\theta_T$  by the ratio of partial configurations  $\|\Theta_T^{(i)}\|/\|\Theta_T\|$  that fall into partition  $i$ .

$$\begin{aligned} \hat{a}_U(\theta_U) &= \frac{1}{\|\Theta_T\|} \int \hat{y}(\theta_{N|U}) d\theta_T \\ &= \frac{1}{\|\Theta_T\|} \int \sum_{P_i \in \mathcal{P}} \mathbb{I}(\theta_{N|U} \in P_i) c_i d\theta_T \end{aligned} \quad (7)$$

$$= \frac{1}{\|\Theta_T\|} \int \sum_{P_i \in \mathcal{P}} \mathbb{I}(\theta_U \in \Theta_U^{(i)}) \mathbb{I}(\theta_T \in \Theta_T^{(i)}) c_i d\theta_T \quad (8)$$

$$= \sum_{P_i \in \mathcal{P}} \mathbb{I}(\theta_U \in \Theta_U^{(i)}) c_i \frac{1}{\|\Theta_T\|} \int \mathbb{I}(\theta_T \in \Theta_T^{(i)}) d\theta_T \quad (9)$$

$$= \sum_{P_i \in \mathcal{P}} \frac{\|\Theta_T^{(i)}\|}{\|\Theta_T\|} \mathbb{I}(\theta_U \in \Theta_U^{(i)}) c_i \quad (10)$$

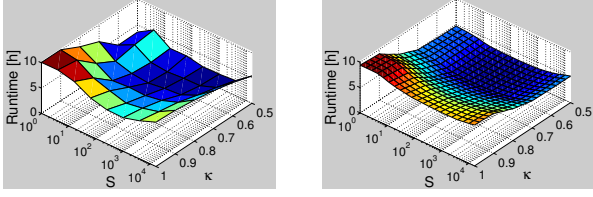
□

### A.2. Proof of Theorem 7

The key idea of the proof is to exploit the fact that, typically, several leaves share the same subdomain of a given hyperparameter. This is because every inner node in the tree only splits one subdomain, leaving all other subdomains unmodified; using pointers to these subdomains thus reduces space complexity. Algorithm 1 only creates new subdomains in Line 4. This line is executed for all  $L - 1$  inner nodes. The sum of the space complexities for the two new sets is in  $O(D)$  since they partition a single hyperparameter’s subdomain. Each of the  $L$  leaf nodes returns its partition  $\Theta^{(i)}$  as a set of  $n$  pointers to the sets  $\Theta_1^{(i)}, \dots, \Theta_n^{(i)}$ . The space complexity is thus  $O(L \cdot D + L \cdot n)$ . □

### A.3. Proof of Corollary 8

As a preprocessing step, we run Algorithm 1 for each of the  $B$  trees in the forest to gather the partitionings  $\mathcal{P}_1, \dots, \mathcal{P}_B$ . We then compute each marginal based on each tree and average results. By Theorem 6, the individual tree marginals are correct, and taking the mean of these marginals is identical to taking the marginals of the mean (since expectation is a linear operation). Time and space complexity for the preprocessing follow directly from Theorem 7. Based on a partitioning  $\mathcal{P}$  with  $L$  leaves, the time complexity for computing a marginal  $\hat{a}_U(\theta_U)$  via Equation 6 is  $O(L \cdot n \log D)$



(a) True interaction effect (on the measured grid) (b) Predicted interaction effect (on a finer grid)

Figure 7. Interaction effect for Online LDA’s hyperparameters  $\kappa$  and  $S$  for predicting runtime.

since for each of the  $L$  partitions  $i$ , we carry out member set queries  $\theta_j \in \Theta_j^{(i)}$  for up to  $n$  categorical hyperparameters, which require  $O(\log D)$  time each (queries for continuous hyperparameters only take time  $O(1)$ ).  $\square$

#### A.4. Proof of Theorem 9

Lines 2 and 3 of Algorithm 2 compute  $\mathbb{V}$  as

$$\sum_{P_i \in \mathcal{P}} \prod_{j \in N} \frac{\|\Theta_j^{(i)}\|}{\|\Theta_j\|} (c_i - \hat{f}_\theta)^2,$$

which, by very similar algebra as in the proof of Theorem 6 equals  $\frac{1}{\|\Theta\|} \int (\hat{y}(\theta) - \hat{f}_\theta)^2 d\theta$ . The remainder of the algorithm uses simple dynamic programming and our efficient marginal predictions from Theorem 6 to compute the fractions  $\mathbb{F}_U$  of variance explained by each hyperparameter subset  $U$  as  $1/\|\Theta_U\| \cdot \left( \sum_{\theta_U \in \Theta_U} \hat{f}_U(\theta_U)^2 \right) / \mathbb{V}$ ; this equals  $1/\|\Theta_U\| \cdot \left( \int \hat{f}_U(\theta_U)^2 d\theta_U \right) / \mathbb{V}$  since all hyperparameters are categorical. The algorithm’s time and space complexity follow from Theorem 7 combined with  $\sum_{k=1}^K \binom{n}{k} \cdot D^k$  marginal predictions. Computing  $\sum_{W \subseteq U} \hat{f}_W(\theta_W)$  takes time  $2^{|U|}$  based on pre-computed values  $\hat{f}_W(\theta_W)$  for  $W \subsetneq U$ .  $\square$

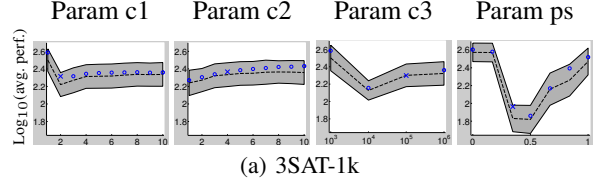
## B. Additional Experimental Data

### B.1. Additional plots for the LDA experiment

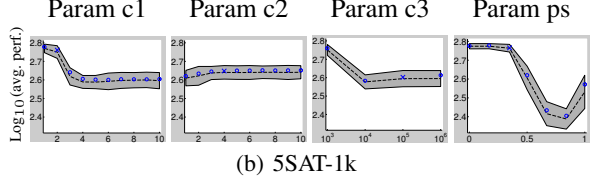
Figure 7 shows the true and predicted interaction effect between  $\kappa$  and  $S$  for the runtime of Online LDA. By inspecting 7(b) in combination with Figure 2(b), we can identify a sweet spot with both low perplexity and runtime at  $S \approx 1000$  and  $\kappa \approx 0.6$ .

### B.2. Details on the AutoWEKA experiment

Table 3 is the full version of Table 1 in the main paper, listing AutoWEKA’s most important hyperparameters for the 21 datasets from Thornton *et al.* (2013), as judged by our functional ANOVA approach based on data from one SMAC run.



(a) 3SAT-1k



(b) 5SAT-1k

Figure 8. Main effects of SPARROW’s 4 hyperparameters on two problem distributions, one plot per hyperparameter (identified by the column header). Each plot shows the marginal performance achieved when varying the hyperparameter’s value across the x-axis. The dashed black line and grey-shaded area indicate predicted marginals (mean  $\pm$  one standard deviation). Blue circles and crosses denote true marginals, with the cross also denoting the default value of each hyperparameter.

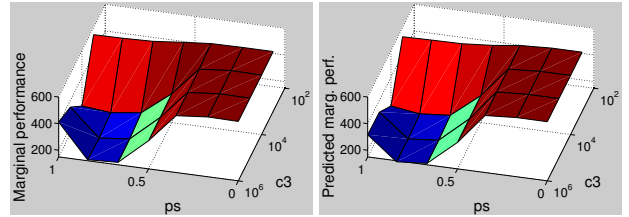


Figure 9. Most important interaction effect between 2 SPARROW hyperparameters on 5SAT500. Left: true marginal performance; right: predicted marginal performance.

Table 4 is the equivalent, but now judged by our functional ANOVA approach based on data from 26 SMAC runs (the same 25 SMAC runs as above and one more; duplicate data points were removed). We note that a smaller fraction of variance is explained by main effects since the additional performance data improves the random forest’s ability to describe interaction effects. We also note that with the additional performance data, on many data sets the model identifies WEKA’s feature selection mechanism as more important than the base classifier to use in an ensemble approach.

### B.3. Evaluation on ground truth data for optimizing a combinatorial problem solver

To verify that our models captured the performance variation of combinatorial problem solvers, we first performed an experiment using ground truth data (much like the Online LDA experiment in Section 4.1). For this purpose, we measured the ground truth performance of all 2800 hyperparameter configurations of the SAT solver SPARROW on each of 250 SAT instances for each of two instance distri-

## An Efficient Approach for Assessing Hyperparameter Importance

Dataset	# data points	time	1st	2nd	3rd
DEXTER	397 ± 284	7 ± 4s	class (53% ± 14%)	baseclass (14% ± 9%)	feature-s (4% ± 6%)
GERMANCREDIT	8676 ± 3154	154 ± 138s	class (36% ± 7%)	baseclass (7% ± 5%)	feature-s (2% ± 1%)
DOROTHEA	239 ± 162	3 ± 2s	class (57% ± 12%)	baseclass (18% ± 12%)	feature-s (3% ± 4%)
YEAST	12823 ± 5194	187 ± 69s	class (31% ± 3%)	feature-s (10% ± 2%)	baseclass (4% ± 2%)
AMAZON	179 ± 154	3 ± 2s	class (58% ± 16%)	baseclass (18% ± 17%)	feature-s (5% ± 4%)
SECOM	862 ± 446	9 ± 5s	class (38% ± 16%)	baseclass (34% ± 18%)	feature-s (1% ± 1%)
SEMEION	1342 ± 1017	22 ± 12s	class (43% ± 9%)	feature-s (9% ± 3%)	baseclass (6% ± 2%)
CAR	8500 ± 4138	134 ± 76s	class (35% ± 3%)	baseclass (7% ± 3%)	feature-s (4% ± 1%)
MADOLON	1768 ± 1124	29 ± 15s	class (45% ± 12%)	baseclass (10% ± 3%)	feature-s (3% ± 2%)
KR-VS-KP	5727 ± 3999	110 ± 64s	class (32% ± 4%)	feature-s (9% ± 2%)	baseclass (7% ± 4%)
ABALONE	1808 ± 1019	35 ± 31s	class (40% ± 13%)	baseclass (8% ± 3%)	feature-s (6% ± 4%)
WINE QUALITY	3934 ± 1834	62 ± 21s	class (33% ± 5%)	baseclass (9% ± 3%)	feature-s (4% ± 2%)
WAVEFORM	2377 ± 1494	41 ± 23s	class (41% ± 7%)	feature-s (11% ± 4%)	baseclass (5% ± 2%)
GISSETTE	153 ± 62	4 ± 2s	class (61% ± 13%)	baseclass (14% ± 8%)	feature-s (4% ± 3%)
CONVEX	173 ± 136	5 ± 4s	class (60% ± 17%)	baseclass (16% ± 12%)	feature-s (5% ± 8%)
CIFAR-10-SMALL	96 ± 64	2 ± 1s	class (56% ± 14%)	baseclass (20% ± 18%)	feature-s (5% ± 6%)
MNIST BASIC	129 ± 62	3 ± 1s	class (55% ± 15%)	baseclass (15% ± 10%)	feature-s (6% ± 5%)
ROT. MNIST + BI	149 ± 147	3 ± 2s	class (57% ± 18%)	baseclass (17% ± 15%)	feature-s (5% ± 4%)
SHUTTLE	743 ± 366	17 ± 7s	class (49% ± 10%)	baseclass (10% ± 5%)	feature-s (3% ± 2%)
KDD09-APPENTENCY	682 ± 404	7 ± 3s	class (41% ± 11%)	baseclass (16% ± 5%)	feature-s (3% ± 3%)
CIFAR-10	99 ± 55	2 ± 1s	class (53% ± 16%)	baseclass (23% ± 15%)	feature-s (4% ± 4%)

Table 3. Top 3 most important hyperparameters of Auto-WEKA for each of 21 datasets, based on data from one SMAC run. We also list the number of data points (*i.e.*, (WEKA configuration, performance) pairs) gathered by SMAC, as well as fANOVA’s runtime using this data. This is the full version of Table 1 in the main paper. For all entries, we give means and standard deviations computed across 25 repetitions (each based on a different SMAC run).

Dataset	# data points	time	1st	2nd	3rd
DEXTER	9954	147s	class (20%)	feature-s (7%)	feature-e (3%)
GERMANCREDIT	236128	3721s	class (37%)	feature-e (2%)	feature-s (2%)
DOROTHEA	4962	66s	class (15%)	baseclass (7%)	feature-e (6%)
YEAST	307076	6957s	class (29%)	feature-s (10%)	feature-e (2%)
AMAZON	4245	54s	class (37%)	baseclass (5%)	feature-s (3%)
SECOM	20856	173s	class (27%)	baseclass (22%)	feature-e (1%)
SEMEION	34379	557s	class (28%)	feature-s (19%)	baseclass (1%)
CAR	255959	3553s	class (38%)	feature-s (4%)	feature-e (2%)
MADOLON	42256	689s	class (35%)	feature-s (3%)	baseclass (3%)
KR-VS-KP	146444	2697s	class (28%)	feature-s (11%)	baseclass (2%)
ABALONE	47205	771s	class (21%)	feature-s (7%)	baseclass (4%)
WINE QUALITY	106260	1585s	class (29%)	baseclass (4%)	feature-s (3%)
WAVEFORM	58917	1002s	class (37%)	feature-s (11%)	baseclass (1%)
GISSETTE	3585	76s	class (23%)	baseclass (7%)	feature-s (6%)
CONVEX	4174	99s	class (26%)	baseclass (9%)	feature-s (2%)
CIFAR-10-SMALL	2418	43s	class (26%)	baseclass (6%)	feature-s (5%)
MNIST BASIC	3072	67s	class (29%)	feature-s (17%)	baseclass (2%)
ROT. MNIST + BI	3293	50s	class (22%)	baseclass (8%)	feature-s (5%)
SHUTTLE	18869	413s	class (35%)	feature-s (4%)	feature-e (2%)
KDD09-APPENTENCY	16833	126s	class (18%)	feature-e (11%)	feature-s (2%)
CIFAR-10	2593	39s	class (28%)	feature-s (3%)	baseclass (3%)

Table 4. Top 3 most important hyperparameters of Auto-WEKA for each of 21 datasets, based on combined data from 26 SMAC runs. We also list the combined number of data points gathered by SMAC (*i.e.*, (WEKA configuration, performance) pairs), as well as fANOVA’s runtime using this data.

butions (a total of  $2 \times 700\,000$  algorithm runs), allowing us to compute actual (as opposed to predicted) marginals across its entire configuration space, as well as arbitrary interaction effects. For each of these instance distributions, the input to our functional ANOVA procedure was the union of performance data collected by SMAC runs and random runs as described in Section B.4 (about 60 000 algorithm runs for each of the two instance distributions, *i.e.*, less than 10% of the algorithm runs required for the ground truth).

For each of the two instance distributions, Figure 8 shows true and predicted main effects of SPARROW’s 4 hyperparameters. Figure 9 shows true and predicted marginals for the most important interaction effect on one distribution (explaining 9.7% of the total variance). As the plot shows, large values of SPARROW’s smoothing probability  $p_s$  and small values of the inverse multiplier for variable age  $c_3$  are generally preferable, but setting both hyperparameters to their respective limits yields poor performance. Several

Problem	Algorithm	#Params	Total #configs	Reference
SAT	SPEAR 1.2.1	26	$8.34 \times 10^{17}$	Babić and Hutter (2007)
SAT	CRYPTOMINISAT 2.9.5	36	$5.0 \times 10^{13}$	Soos (2010)
SAT	SPARROW	4	2800	Balint <i>et al.</i> (2011)
SAT	CAPTAINJACK	33	$4.98 \times 10^{30}$	Tompkins <i>et al.</i> (2011)
SAT	SATENSTEIN	51	$1.38 \times 10^{34}$	KhudaBukhsh <i>et al.</i> (2009)
MIP	CPLEX 12.1	76	$1.90 \times 10^{47}$	IBM Corp. (2014)
ASP	CLASP 2.1	85	$1.42 \times 10^{49}$	Gebser <i>et al.</i> (2007)

Table 5. Combinatorial problem solvers and their hyperparameter configuration spaces.

Problem	Benchmark	Application	Reference
SAT	BMC	Bounded model checking	Zarpas (2005)
SAT	SWV	Software verification	Babić and Hu (2007)
SAT	3SAT1k	Unif. random 3-SAT	Tompkins <i>et al.</i> (2011)
SAT	5SAT500	Unif. random 5-SAT	Tompkins <i>et al.</i> (2011)
MIP	RCW	Wildlife conservation	Ahmadizadeh <i>et al.</i> (2010)
MIP	CORLAT	Wildlife conservation	Gomes <i>et al.</i> (2008)
MIP	Regions200	Combinatorial auctions	Leyton-Brown <i>et al.</i> (2000)
MIP	CLS	Capacitated lot-sizing	Atamtürk and Muñoz (2004)
ASP	WeightSeq	Database query optimization	Silverthorn <i>et al.</i> (2012)
ASP	Riposte	Software Verification	Silverthorn <i>et al.</i> (2012)

Table 6. Instance distributions.

designers of algorithms for combinatorial problems have remarked that detecting and visualizing important interaction effects in this way dramatically facilitates their task of understanding their algorithm’s empirical performance.

#### B.4. Details on Experiments with Combinatorial Problem Solvers

We performed experiments with a set of standard algorithm configuration scenarios from the literature. Tables 5 and 6 summarize the solvers and benchmark instance distributions we used in our experiments with combinatorial problems. We used the industrial SAT solvers SPEAR and CRYPTOMINISAT on the SAT-encoded formal verification benchmarks BMC and SWV; the stochastic local search solvers SPARROW, CAPTAINJACK and SATENSTEIN on two uniform random satisfiable  $k$ -SAT benchmarks; the industrial MIP solver CPLEX on four distributions from industry and computational sustainability; and the ASP solver CLASP on database query optimization and software verification benchmarks.

These combinations of solvers and benchmarks resulted in the 16 configuration scenarios listed in Table 7. For each of these configuration scenarios, we gathered algorithm performance data using 10 independent runs of the configuration procedure SMAC (Hutter *et al.*, 2011) with adaptive capping (Hutter *et al.*, 2013a). Table 7 provides details on these SMAC runs and their performance. Briefly, the ten SMAC runs carried out between 13452 and 454336 solver runs and resulted in speedups between 1.02 and 857 times compared to the solvers’ default configurations. Table 7 provides details for the solver/benchmark combinations and on the result of configuration with SMAC. The speedups in penalized average runtime obtained by SMAC range between a factor of 1.02 (CRYPTOMINISAT-BMC) and 857 (SATENSTEIN-5SAT500).

Bayesian optimization methods like SMAC focus their efforts on high-performance regions and do not much explore poor regions of the hyperparameter space. Using adaptive capping also results in many right-censored runs for which only a (relatively small) upper bound on algorithm runtime is available. Thus, from this data alone it is hard to achieve realistic runtime predictions for poor configurations. In order to nevertheless achieve some global coverage, following Hutter *et al.* (2013b), we also added 10000 random solver runs (each using one random configuration and one random instance). While in the main paper we used models trained on the union of the SMAC runs and the 10000 random runs, in Table 8 and 9, we report equivalent results for models trained only on data from either source, with quite similar results as in the main paper.

#### B.5. Details on Subspace Configuration Experiments

We performed subspace configuration experiments for both a machine learning algorithm (a neural network) and a combinatorial problem solver (CPLEX). For the former, we used the neural network implementation of WEKA (which has 8 hyperparameters) and several datasets taken from AutoWEKA (Thornton *et al.*, 2013). For each dataset, we executed 10 SMAC runs for one day to optimize the neural network’s hyperparameters, using AutoWEKA’s standard of cutting off unsuccessful algorithm runs after one hour of CPU time or if it used more than 2 GB of RAM. For each dataset, we then used the union of the logged performed data from these 10 SMAC runs as an input to functional ANOVA to determine the three neural network hyperparameters with the largest main effect for the dataset (using “improvement over the 25% quantile” as described in the main paper). We then ran another 10 SMAC runs in this hyperparameter subspace that were otherwise identical to the 10 SMAC runs in the full space. As Figure 10 illustrates, the subspace configuration experiments often found very good configurations and made progress faster than configuration in the full space. Only in one case (MNIST) they did not find as good a configuration as a search in the full space.

To assess the true importance of the selected hyperparameters for the combinatorial problem solver CPLEX, we used not only SMAC but also another state-of-the-art configuration procedure, ParamILS (Hutter *et al.*, 2009). As input data to functional ANOVA, we used the union of SMAC runs and random runs as described in Section B.4. We selected the 10 CPLEX hyperparameters (out of 76) with the largest main effects (using “improvement over the 25% quantile” as described in the main paper) and executed another 10 runs of both ParamILS and SMAC on the resulting subspace. As shown in Figure 11, both SMAC and ParamILS found good configurations in the reduced CPLEX spaces. Their initial progress was faster in the subspace than in the full space while for longer configuration budgets also optimizing less important hyperparameters yielded further progress.

Scenario	Captime per algorithm run	Configuration budget	Penalized avg. runtime			#Runs executed by $10 \times$ SMAC
			Default	SMAC	Speedup factor	
SPEAR-BMC	300s	4 days	928s	864s	10.7 $\times$	44 964
SPEAR-SWV	300s	1 day	559s	1s	559 $\times$	442 238
CRYPTOMINISAT-BMC	300s	4 days	761s	747s	1.02 $\times$	42 511
CRYPTOMINISAT-SWV	300s	1 day	5s	2s	2.5 $\times$	244 845
SPARROW-3SAT1k	600s	1 day	162s	80s	2.0 $\times$	50 548
SPARROW-5SAT500	600s	1 day	5 978s	16s	374 $\times$	43 240
CAPTAINJACK-3SAT1k	600s	1 day	3 269s	108s	30 $\times$	38 799
CAPTAINJACK-5SAT500	600s	1 day	6 000s	23s	261 $\times$	13 452
SATENSTEIN-3SAT1k	600s	1 day	609s	56s	11 $\times$	45 665
SATENSTEIN-5SAT500	600s	1 day	6 000s	7s	857 $\times$	105 272
CPLEX-RCW	10 000s	4 days	171s	92s	1.9 $\times$	28 980
CPLEX-CORLAT	10 000s	4 days	815s	6s	136 $\times$	269 611
CPLEX-Regions200	10 000s	4 days	28s	5s	5.6 $\times$	382 823
CPLEX-CLS	10 000s	4 days	24s	5s	4.8 $\times$	345 688
CLASP-WeightedSeq	900s	4 days	2 981s	831s	3.6 $\times$	25 822
CLASP-Riposte	300s	1 day	38s	1s	38 $\times$	454 336

Table 7. Details on the SMAC runs on all solver/benchmark combinations. Captime denotes the time after which a single algorithm run is terminated if not successful yet; configuration budget is the CPU time allocated to each SMAC run; penalized average runtime of the default configuration and configurations found by SMAC is the average runtime, counting runs timed-out at the captime  $\kappa_{max}$  as  $10 \times \kappa_{max}$ .

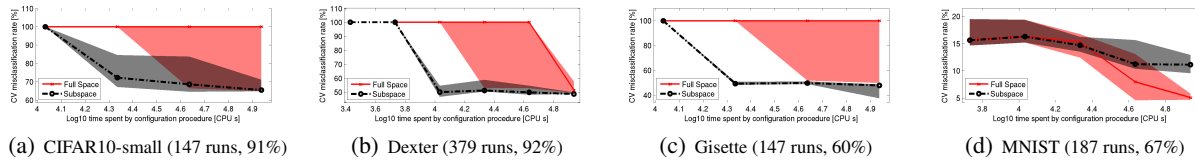


Figure 10. Algorithm configuration with SMAC for the full and reduced parameter configuration spaces of the WEKA implementation of neural networks for 4 of the AutoWEKA datasets. The subcaptions give the number of runs gathered in the 10 independent SMAC runs on the dataset, which were used as input to the functional approach for identifying the important subspace. The runtime of functional ANOVA was below one second in each case.

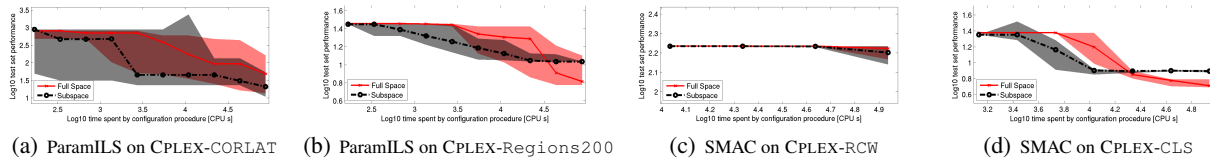


Figure 11. Algorithm configuration results for ParamILS and SMAC on full and reduced parameter configuration spaces of CPLEX.

**An Efficient Approach for Assessing Hyperparameter Importance**

Scenario	Raw Performance		Improvement over q25		Improvement over def	
	Main	Pairwise	Main	Pairwise	Main	Pairwise
SPEAR-BMC	90% (1s)	4% (70s)	51% (0s)	18% (20s)	47% (0s)	19% (18s)
SPEAR-SWV	76% (1s)	11% (72s)	61% (0s)	19% (26s)	66% (1s)	16% (38s)
CRYPTOMINISAT-BMC	37% (1s)	30% (42s)	29% (1s)	18% (28s)	28% (1s)	20% (28s)
CRYPTOMINISAT-SWV	35% (1s)	29% (44s)	24% (0s)	25% (21s)	22% (0s)	26% (18s)
SPARROW-3SAT1k	74% (0s)	19% (0s)	40% (0s)	35% (0s)	19% (0s)	36% (0s)
SPARROW-5SAT500	65% (0s)	28% (0s)	56% (0s)	35% (0s)	65% (0s)	28% (0s)
CAPTAINJACK-3SAT1k	52% (3s)	13% (490s)	17% (2s)	12% (245s)	22% (2s)	12% (253s)
CAPTAINJACK-5SAT500	34% (3s)	22% (383s)	26% (1s)	22% (173s)	32% (2s)	22% (356s)
SATENSTEIN-3SAT1k	41% (3s)	41% (355s)	38% (1s)	31% (135s)	25% (1s)	26% (90s)
SATENSTEIN-5SAT500	15% (2s)	61% (297s)	8% (1s)	64% (119s)	7% (1s)	52% (75s)
CPLEX-RCW	73% (1s)	13% (110s)	40% (0s)	26% (44s)	35% (0s)	26% (39s)
CPLEX-CORLAT	32% (1s)	28% (114s)	28% (0s)	27% (47s)	25% (0s)	28% (56s)
CPLEX-Regions200	34% (1s)	23% (108s)	40% (0s)	30% (52s)	20% (0s)	34% (25s)
CPLEX-CLS	28% (1s)	23% (111s)	21% (0s)	25% (53s)	18% (0s)	30% (40s)
CLASP-WeightedSeq	60% (9s)	12% (1532s)	34% (4s)	17% (670s)	30% (3s)	16% (570s)
CLASP-Riposte	75% (7s)	13% (1154s)	14% (5s)	6% (834s)	23% (3s)	5% (529s)

Table 8. Same as Table 1 in the main paper, but based on a model trained on 10 000 random data points only (instead of the union of these and the data points logged in 25 SMAC runs).

Scenario	Raw Performance		Improvement over q25		Improvement over def	
	Main	Pairwise	Main	Pairwise	Main	Pairwise
SPEAR-BMC	79% (1s)	6% (45s)	50% (0s)	15% (16s)	22% (0s)	21% (10s)
SPEAR-SWV	37% (5s)	12% (277s)	14% (1s)	14% (72s)	39% (4s)	11% (267s)
CRYPTOMINISAT-BMC	24% (1s)	17% (20s)	28% (0s)	22% (12s)	7% (0s)	15% (2s)
CRYPTOMINISAT-SWV	33% (3s)	24% (133s)	5% (1s)	11% (39s)	24% (1s)	37% (58s)
SPARROW-3SAT1k	82% (0s)	12% (0s)	52% (0s)	26% (0s)	36% (0s)	32% (0s)
SPARROW-5SAT500	83% (0s)	14% (0s)	50% (0s)	34% (0s)	84% (0s)	14% (0s)
CAPTAINJACK-3SAT1k	10% (5s)	13% (649s)	25% (5s)	14% (650s)	41% (4s)	10% (647s)
CAPTAINJACK-5SAT500	8% (2s)	16% (264s)	14% (2s)	15% (249s)	10% (2s)	16% (235s)
SATENSTEIN-3SAT1k	47% (3s)	29% (470s)	47% (3s)	28% (448s)	39% (3s)	27% (349s)
SATENSTEIN-5SAT500	67% (6s)	21% (765s)	56% (4s)	31% (492s)	64% (6s)	21% (756s)
CPLEX-RCW	52% (4s)	20% (597s)	54% (3s)	12% (516s)	9% (1s)	16% (145s)
CPLEX-CORLAT	17% (27s)	11% (4152s)	27% (14s)	11% (2046s)	21% (24s)	23% (3749s)
CPLEX-Regions200	74% (70s)	12% (10216s)	70% (62s)	12% (9035s)	39% (50s)	10% (7395s)
CPLEX-CLS	18% (143s)	5% (21062s)	2% (43s)	4% (5909s)	14% (91s)	3% (13356s)
CLASP-WeightedSeq	12% (4s)	12% (771s)	24% (3s)	14% (519s)	25% (3s)	18% (563s)
CLASP-Riposte	10% (94s)	2% (16696s)	9% (67s)	3% (11680s)	15% (75s)	3% (13359s)

Table 9. Same as Table 2 in the main paper, but based on a model trained on the data points logged in 25 SMAC runs only (instead of the union of these and 10 000 random data points).