



Active Learning for SAT Solver Benchmarking

Tobias Fuchs^(✉), Jakob Bach, and Ashlin Iser

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
`info@tobiasfuchs.de, {jakob.bach,ashlin.iser}@kit.edu`

Abstract. Benchmarking is a crucial phase when developing algorithms. This also applies to solvers for the SAT (propositional satisfiability) problem. Benchmark selection is about choosing representative problem instances that reliably discriminate solvers based on their runtime. In this paper, we present a dynamic benchmark selection approach based on active learning. Our approach predicts the rank of a new solver among its competitors with minimum runtime and maximum rank prediction accuracy. We evaluated this approach on the Anniversary Track dataset from the 2022 SAT Competition. Our selection approach can predict the rank of a new solver after about 10 % of the time it would take to run the solver on all instances of this dataset, with a prediction accuracy of about 92 %. We also discuss the importance of instance families in the selection process. Overall, our tool provides a reliable way for solver engineers to determine a new solver’s performance efficiently.

Keywords: Propositional satisfiability · Benchmarking · Active learning

1 Introduction

One of the main phases of algorithm engineering is benchmarking. This also applies to propositional satisfiability (SAT), the archetypal $\mathcal{NP}$ -complete problem. Benchmarking is, however, quite expensive regarding the runtime of experiments. While benchmarking a single SAT solver might still be feasible, developing new, competitive SAT solvers requires extensive experimentation with a variety of ideas [8,2]. In particular, a new solver idea is rarely best on the first try. Thus, it is highly desirable to reduce benchmarking time and discard unpromising ideas early, allowing to test more approaches or spend more time on promising ones. The field of SAT solver benchmarking is well established, but traditional benchmark selection approaches do not optimize benchmark runtime. Instead, they focus on selecting a representative set of instances for scoring solvers [10,15]. For the latter, SAT Competitions typically employ the PAR-2 score, i.e., the average runtime with a penalty of 2τ for timeouts with time-limit τ [8].

In this paper, we present a novel benchmark selection approach based on active learning. Our approach can predict the rank of a new solver with high accuracy in only a fraction of the time needed to evaluate the complete benchmark. Definition 1 specifies the problem we address.

Definition 1 (New-Solver Problem). *Given solvers $\mathcal{A}$, instances $\mathcal{I}$, runtimes $r: \mathcal{A} \times \mathcal{I} \rightarrow [0, \tau]$ with time-limit τ , and a new solver $\hat{a} \notin \mathcal{A}$, incrementally select benchmark instances from $\mathcal{I}$ to maximize the confidence in predicting the rank of $\hat{a}$ while minimizing the total benchmark runtime.*

Note that our scenario assumes knowing the runtimes of all solvers, except the new one, on all instances. One could also imagine a collaborative filtering scenario, where runtimes are only partially known [23,25].

Our approach satisfies several desirable criteria for benchmarking: Rather than outputting a binary classification, i.e., whether the new solver is worse than an existing solver or not, we provide a *scoring* function that shows by which margin a solver is worse and how similar it is to existing solvers. In particular, our approach enables *ranking* the new solver amidst a set of existing solvers. For this ranking, we do not even need to predict exact solver runtimes, which is trickier. Further, we optimize the *runtime* that our strategy needs to arrive at its conclusion. We use instance and runtime *features*. Moreover, we select instances *non-randomly* and *incrementally*. In particular, we consider runtime information from already done experiments when choosing the next. By doing so, we can control the properties of the benchmarking approach, such as its required runtime. Our approach is *scalable* in that it ranks a new solver $\hat{a}$ among any number of known solvers $\mathcal{A}$. In particular, we only subsample the benchmark once instead of comparing pairwise against each other solver [21].

We evaluate our approach with the SAT Competition 2022 Anniversary Track dataset [2], consisting of 5355 instances and runtimes of 28 solvers. We perform cross-validation by treating each solver once as the new solver and learning to predict the PAR-2 rank of that solver. On average, our predictions reach about 92 % accuracy with only about 10 % of the runtime required to evaluate these solvers on the complete set of instances.

Our entire source code¹ and experimental data² are available on GitHub.

2 Related Work

Benchmarking is not only of high interest in many fields but also an active research area on its own. Recent studies show that benchmark selection is challenging for multiple reasons. Biased benchmarks can easily lead to fallacious interpretations [7]. Benchmarking also has many interchangeable parts, such as the performance measures used, how measurement points are aggregated, and how missing values are handled. Questionable research practices could alter these elements a-posteriori to meet expectations, thereby skewing the results [27]. In the following, we discuss related work from the areas of static benchmark selection, algorithm configuration, incremental benchmark selection, and active learning. Table 1 compares the most relevant approaches, which all pursue slightly different goals. Thus, our approach is *not* a general improvement over the others but the only one fully aligned with Definition 1.

¹ <https://github.com/mathefuchs/al-for-sat-solver-benchmarking>

² <https://github.com/mathefuchs/al-for-sat-solver-benchmarking-data>

Table 1: Comparison of features of our benchmark-selection approach, the static benchmark-selection approach by Hoos et al. [15], the algorithm configuration system SMAC [16], and the active-learning approaches by Matricon et al. [21].

Feature	Hoos [15]	SMAC [16]	Matricon [21]	Our approach
Ranking/Scoring	✓	✗	(✓)	✓
Runtime Minimization	✗	✓	✓	✓
Incremental/Non-Random	✗	✗	✓	✓
Scalability	✓	✓	✗	✓

Static Benchmark Selection. Benchmark selection is essential for competitions, e.g., the SAT Competition. In such competitions, the organizers define the rules for composing the benchmarks. These selection strategies are primarily static, i.e., they do not depend on particular solvers to distinguish. Balint et al. provide an overview of benchmark-selection criteria in different solver competitions [1]. Froleyks et al. describe benchmark selection in recent SAT competitions [8]. Manthey and Möhle find that competition benchmarks might contain redundant instances and propose a feature-based approach to remove redundancy [20]. Misir presents a feature-based approach to reduce benchmarks by matrix factorization and clustering [24].

Hoos et al. [15] discuss which properties are most desirable when selecting SAT benchmark instances. The selection criteria are instance variety to avoid over-fitting, adapted instance hardness (not too easy but also not too hard), and avoiding duplicate instances. To filter too similar instances, they use a distance-based approach with the SATzilla features [37,38]. The approach does, however, not optimize for benchmark *runtime* and selects instances *randomly*, apart from constraints on the instance hardness and feature distance.

Algorithm Configuration. Further related work can be found within the field of algorithm configuration [14,32], e.g., the configuration system SMAC [16]. Thereby, the goal is to tune SAT solvers for a given sub-domain of problem instances. Although this task is different from our goal, e.g., we do not need to navigate the configuration space, there are similarities to our approach as well. For example, SMAC also employs an iterative, model-based selection procedure, though for configurations rather than instances. An algorithm configurator, however, cannot be used to *rank/score* a new solver since algorithm configuration solemnly seeks to find the best-performing configuration. Also, while using a model-based selection strategy to sample configurations, instance selection is made *randomly*, i.e., without building a model over instances.

Incremental Benchmark Selection. Matricon et al. present an incremental benchmark selection approach [21]. Their *per-set efficient algorithm selection problem* (PSEAS) is similar to our *New-Solver Problem* (cf. Definition 1). Given a pair of SAT solvers, they iteratively select a subset of instances until the

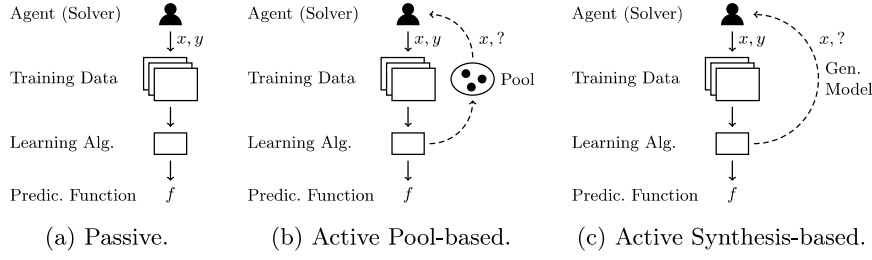


Fig. 1: Types of machine learning (depiction inspired by Rubens et.al. [29]).

desired confidence level is reached to decide which of the two solvers is better. The selection of instances depends on the choice of the solvers to distinguish. They calculate a scoring metric for all unselected instances, run the experiment with the highest score, and update the confidence. Their approach ticks off most of our desired features in Table 1. However, the approach only compares solvers binarily rather than providing a *scoring*. Thus, it is unclear how similar two given solvers are or on which instances they behave similarly. Moreover, a significant shortcoming is the lacking *scalability* with the number of solvers. Comparing only pairs of solvers, evaluating a new solver requires sampling a separate benchmark for each existing solver. In contrast, our approach allows comparing a new solver against a set of existing solvers by sampling only one benchmark.

Active Learning. Prediction models in passive machine learning are trained on datasets with given instance labels (cf. Fig. 1a). In contrast, active learning (AL) starts with no or little labeled data. It repeatedly selects interesting problem instances for which to acquire labels, aiming to gradually improve the prediction model (cf. Fig. 1b). AL methods are especially beneficial if acquiring labels is computationally expensive, like obtaining solver runtimes. Without AL methods, it is not obvious which instances to label and which not. On the one hand, we want to maximize the utility an instance provides to our model, i.e., rank prediction accuracy, and on the other hand, minimize the cost, i.e., predicted runtime, associated with the instance’s acquisition. Thus, we strive for an accurate prediction model without having to label every data point.

Rubens et. al. [29] survey active-learning advances. While synthesis-based AL methods [5,9,34] generate instances for labeling, pool-based methods [11,13,19] rely on a fixed set of unlabeled instances to sample from. Recent synthesis-based methods within the field of SAT solving show how to generate problem instances with desired properties [5,9]. This goal is, however, orthogonal to ours. While those approaches want to generate instances on which a solver is good or bad, we want to predict whether a solver is good or bad on an existing benchmark. Volpato and Guangyan use pool-based AL to learn an instance-specific algorithm selector [35]. Rather than benchmarking a solver’s overall performance, their goal is to recommend the best solver out of a set of solvers for each SAT instance.

Algorithm 1: Incremental Benchmarking Framework**Input:** Solvers $\mathcal{A}$, Instances $\mathcal{I}$, Runtimes $r : \mathcal{A} \times \mathcal{I} \rightarrow [0, \tau]$, Solver $\hat{a}$ **Output:** Predicted Score of $\hat{a}$, Measured Runtimes $\mathcal{R}$

```

1  $\mathcal{M} \leftarrow \text{initModel}(\mathcal{A}, \mathcal{I}, r, \hat{a})$  // cf. Section 3.1
2  $\mathcal{R} \leftarrow \emptyset$ 
3 while not stop( $\mathcal{M}$ ) do // cf. Section 3.3
4    $e \leftarrow \text{selectNextInstance}(\mathcal{M})$  // cf. Section 3.2
5    $t \leftarrow \text{runExperiment}(\hat{a}, e)$  // Runs  $\hat{a}$  on  $e$  with timeout  $\tau$ 
6    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(e, t)\}$ 
7    $\text{updateModel}(\mathcal{M}, \mathcal{R})$  // cf. Section 3.1
8  $s_{\hat{a}} \leftarrow \text{predictScore}(\mathcal{M})$  // cf. Section 3.1
9 return ( $s_{\hat{a}}, \mathcal{R}$ )
```

3 Active Learning for SAT Solver Benchmarking

Algorithm 1 outlines our benchmarking framework. Given a set of solvers $\mathcal{A}$, instances $\mathcal{I}$ and runtimes r , we first initialize a prediction model $\mathcal{M}$ for the new solver $\hat{a} \notin \mathcal{A}$ (Line 1). The prediction model $\mathcal{M}$ is used to repeatedly select an instance (Line 4) for benchmarking $\hat{a}$ (Line 5). The acquired result is subsequently used to update the prediction model $\mathcal{M}$ (Line 7). When the stopping criterion is met (Line 3), we quit the benchmarking loop and predict the final score of $\hat{a}$ (Line 8). Algorithm 1 returns the predicted score of $\hat{a}$ as well as the acquired instances and runtime measurements (Line 9).

Section 3.1 describes the underlying prediction model $\mathcal{M}$ and specifies how we may derive a solver ranking from it. We discuss criteria for selecting instances in Section 3.2. Section 3.3 concludes with possible stopping conditions.

3.1 Solver Model

The model M provides a runtime-label prediction function $f : \hat{\mathcal{A}} \times \mathcal{I} \rightarrow \mathbb{R}$ for all solvers $\hat{\mathcal{A}} := \mathcal{A} \cup \{\hat{a}\}$. This prediction function powers instance selection as described in Section 3.2. During model updates (Algorithm 1, Line 7), f is trained to predict a transformed version of the acquired runtimes $\mathcal{R}$. We describe the runtime transformation in the subsequent section. The features described in Section 4.2 serve as the input to the model. Further, note that we build a new prediction model in each iteration since running experiments (Line 5) dominates the runtime of model training by magnitudes. Finally, we predict the score of the new solver $\hat{a}$ with the prediction function f (Line 8).

Runtime Transformation. For the prediction model M , we transform the real-valued runtimes into discrete runtime labels on a per-instance basis. For each instance $e \in \mathcal{I}$, we use a clustering algorithm to assign the runtimes in $\{r(a, e) \mid a \in \mathcal{A}\}$ to one of k clusters $C_1, \dots, C_k$ such that the fastest runtimes

for the instance e are in cluster C_1 and the slowest are in cluster C_{k-1} . Timeouts τ always form a separate cluster C_k . The runtime transformation function $\gamma_k : \mathcal{A} \times \mathcal{I} \rightarrow \{1, \dots, k\}$ is then specified as follows:

$$\gamma_k(a, e) = j \Leftrightarrow r(a, e) \in C_j$$

Given an instance $e \in \mathcal{I}$, a solver $a \in \mathcal{A}$ belongs to the $\gamma_k(a, e)$ -fastest solvers on instance e . In preliminary experiments, we achieved higher accuracy for predicting such discrete runtime labels than for predicting raw runtimes. Research on portfolio solvers has also shown that discretization works well in practice [4,26].

Ranking Solvers. To determine solver ranks, we use the transformed runtimes $\gamma_k(a, e)$ in the adapted scoring function $s_k : \mathcal{A} \rightarrow [1, 2 \cdot k]$ as follows:

$$s_k(a) := \frac{1}{|\mathcal{I}|} \sum_{e \in \mathcal{I}} \gamma'_k(a, e) \quad \gamma'_k(a, e) := \begin{cases} 2 \cdot \gamma_k(a, e) & \text{if } \gamma_k(a, e) = k \\ \gamma_k(a, e) & \text{otherwise} \end{cases} \quad (1)$$

I.e., we apply PAR-2 scoring, which is commonly used in SAT competitions [8], on the discrete labels. The scoring function s_k induces a ranking among solvers.

3.2 Instance Selection

Selecting an instance based on the model is a core functionality of our framework (cf. Algorithm 1, Line 4). In this section, we introduce two instance sampling strategies, one that minimizes uncertainty and one that maximizes information gain. Both strategies use the model's label-prediction function f and are inspired by existing work within the realms of active learning [30]. These methods require the model's predictions to include probabilities for the k discrete runtime labels. Let $f' : \hat{\mathcal{A}} \times \mathcal{I} \rightarrow [0, 1]^k$ denote this modified prediction function. In the following, the set $\tilde{\mathcal{I}} \subseteq \mathcal{I}$ denotes the instances that have already been sampled.

Uncertainty Sampling. The uncertainty sampling strategy selects the instance closest to the model's decision boundary, i.e., we select the instance $e \in \mathcal{I} \setminus \tilde{\mathcal{I}}$ that minimizes $U(e)$, which is specified as follows:

$$U(e) := \left| \frac{1}{k} - \max_{n \in \{1, \dots, k\}} f'(\hat{a}, e)_n \right|$$

Information-Gain Sampling. The information-gain sampling strategy selects the instance with the highest expected entropy reduction regarding the runtime labels of the instance. To be more specific, we select the instance $e \in \mathcal{I} \setminus \tilde{\mathcal{I}}$ that maximizes $IG(e)$, which is specified as follows:

$$IG(e) := H(e) - \sum_{n=1}^k f'(\hat{a}, e)_n \hat{H}_n(e)$$

Here, $H(e)$ denotes the entropy of the runtime labels $\gamma(a, e)$ over all $a \in \mathcal{A}$ and $H(e, n)$ denotes the entropy of these labels plus n as the runtime label for $\hat{a}$. The term $\hat{H}_n(e)$ is computed for every possible runtime label $n \in \{1, \dots, k\}$. By maximizing information gain, we select instances that identify solvers with similar behavior.

3.3 Stopping Criteria

In this section, we present the two dynamic stopping criteria in our experiments, the Wilcoxon and the ranking stopping criterion (cf. Algorithm 1, Line 3).

Wilcoxon Stopping Criterion. The Wilcoxon stopping criterion stops the active-learning process when we are confident enough that the predicted runtime labels of the new solver are sufficiently different from existing solvers. This criterion is loosely inspired by Matricon et. al. [21]. We use the average p -value $W_{\hat{a}}$ of a Wilcoxon signed-rank test $w(S, P)$ of the two runtime label distributions $S = \{\gamma(a, e) \mid e \in \mathcal{I}\}$ for an existing solver a and $P = \{f(\hat{a}, e) \mid e \in \mathcal{I}\}$ for the new solver $\hat{a}$:

$$W_{\hat{a}} := \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} w(S, P)$$

To improve the stability of this criterion, we use an exponential moving average to smooth out outliers and stop as soon as $W_{\text{exp}}^{(i)}$ drops below a fixed threshold:

$$\begin{aligned} W_{\text{exp}}^{(0)} &:= 1 \\ W_{\text{exp}}^{(i)} &:= \beta W_{\hat{a}} + (1 - \beta) W_{\text{exp}}^{(i-1)} \end{aligned}$$

Ranking Stopping Criterion. The ranking stopping criterion is less sophisticated in comparison. It stops the active-learning process if the ranking induced by the model's predictions (Equation 1) remained unchanged within the last l iterations. However, the concrete values of the predicted score $s_{\hat{a}}$ might still change. We are solemnly interested in the induced ranking in this case.

4 Experimental Design

Given all the previously presented instantiations for Algorithm 1, this section outlines our experimental design, including our evaluation framework, used data sets, hyper-parameter choices, and implementation details.

4.1 Evaluation Framework

As stated in the Introduction, this work addresses the *New-Solver Problem* (cf. Definition 1). As described in Section 3.1, a prediction model $\mathcal{M}$ provides us with an estimated scoring $s_{\hat{a}}$ for the new solver $\hat{a}$.

Algorithm 2: Evaluation Framework

Input: Solvers $\mathcal{A}$, Instances $\mathcal{I}$, Runtimes $r : \mathcal{A} \times \mathcal{I} \rightarrow [0, \tau]$
Output: Average Ranking Accuracy $\bar{O}_{\text{acc}}$, Average Fraction of Runtime $\bar{O}_{\text{rt}}$

```

1  $O \leftarrow \emptyset$ 
2 for  $\hat{a} \in \mathcal{A}$  do
3    $\mathcal{A}' \leftarrow \mathcal{A} \setminus \{\hat{a}\}$ 
4    $(s_{\hat{a}}, \mathcal{R}) \leftarrow \text{runALAlgorithm}(\mathcal{A}', \mathcal{I}, r, \hat{a})$            // Refer to Algorithm 1
   // Determine Ranking Accuracy
5    $O_{\text{acc}} \leftarrow 0$ 
6   for  $a \in \mathcal{A}$  do
7     if  $(s_k(a) - s_{\hat{a}}) \cdot (\text{par}_2(a) - \text{par}_2(\hat{a})) > 0$  then
8        $O_{\text{acc}} \leftarrow O_{\text{acc}} + \frac{1}{|\mathcal{A}|}$ 
   // Determine Runtime Fraction
9    $r \leftarrow \sum_{e \in \mathcal{I}} r(\hat{a}, e)$ 
10   $O_{\text{rt}} \leftarrow 0$ 
11  for  $e \in \mathcal{I}$  do
12    if  $\exists t, (e, t) \in \mathcal{R}$  then
13       $O_{\text{rt}} \leftarrow O_{\text{rt}} + \frac{t}{r}$ 
14   $O \leftarrow O \cup \{(O_{\text{acc}}, O_{\text{rt}})\}$ 
15  $(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}}) \leftarrow \text{average}(O)$ 
16 return  $(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}})$ 

```

To evaluate a concrete instantiation of Algorithm 1, i.e., a concrete choice for all the sub-routines, we perform cross-validation on our set of solvers. Algorithm 2 shows this. That means each solver plays the role of the new solver $\hat{a}$ once (Line 2). Note that the *new* solver in each iteration is excluded from the set of solvers $\mathcal{A}$ to avoid data leakage (Line 3). After running our active-learning framework for solver $\hat{a}$ (Line 4), we compute the value of both our optimization goals, i.e., ranking accuracy and runtime. We define the *ranking accuracy* $O_{\text{acc}} \in [0, 1]$ (higher is better) by the fraction of pairs $(\hat{a}, a)$ for all $a \in \mathcal{A}$ that are decided correctly regarding the ground-truth scoring par_2 (Lines 5-8). The *fraction of runtime* that the algorithm needs to arrive at its conclusion is denoted by $O_{\text{rt}} \in [0, 1]$ (lower is better). This metric puts the runtime summed over the sampled instances in relation to the runtime summed over all instances in the dataset (Lines 9-13). Finally, we compute averages of the output metrics in Line 15 after we have collected all cross-validation results in Line 14. Overall, we want to find an approach that maximizes

$$O_{\delta} := \delta O_{\text{acc}} + (1 - \delta) (1 - O_{\text{rt}}) \quad , \quad (2)$$

whereby $\delta \in [0, 1]$ allows for linear weighting between the two optimization goals O_{acc} and O_{rt} . Plotting the approaches that maximize O_{δ} for all $\delta \in [0, 1]$ on