Active Learning for SAT Solver Benchmarking

(extended and revised version)

Tobias Fuchs[1*], Jakob Bach[1] and Markus Iser[1]

[1]Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany.

*Corresponding author(s). E-mail(s): tobias.fuchs@kit.edu;
Contributing authors: jakob.bach@kit.edu; markus.iser@kit.edu;

Abstract

Benchmarking is crucial for developing new algorithms. This also applies to solvers for the propositional satisfiability (SAT) 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 estimates the rank of a new solver among its competitors, striving to minimize benchmarking runtime but maximize ranking accuracy. Instead of using real-valued solver runtimes, our approach works with discretized runtime labels, which yielded better solver rank predictions. We evaluated this approach on the Anniversary Track dataset from the SAT Competition 2022. Our benchmark selection approach can predict the rank of a new solver after approximately 10 % of the time it would take to run the solver on all instances of this dataset, with a prediction accuracy of approximately 92 %. Additionally, we discuss the importance of instance families in the selection process. In conclusion, our tool offers a reliable method for solver engineers to assess a new solver's performance efficiently.

Keywords: Propositional satisfiability, Benchmark selection, Active learning

1 Introduction

One of the main phases of algorithm engineering is benchmarking. This also applies to solvers for propositional satisfiability (SAT), the canonical $\mathcal{NP}$-complete problem. Benchmarking is, however, quite expensive regarding the runtime of experiments. While it is still feasible to benchmark a single or a small number of given SAT solvers, developing new, competitive SAT solvers requires extensive experimentation with a

variety of ideas. The latter often results in a combinatorial explosion of the configuration space [18]. 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, yet traditional benchmark selection approaches do not optimize benchmark runtime. Instead, they focus on selecting a representative set of instances for ranking solvers [12, 16]. In this regard, SAT Competitions typically employ the PAR-2 score, which, given a time limit of τ, is the average solver runtime with a penalty of 2τ for timeouts [9].

Problem statement. 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. 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 [26, 28].

Definition 1 (New-Solver Problem)**.** *Let us consider a set of solvers $\mathcal{A}$, instances $\mathcal{I}$, and runtimes $r\colon \mathcal{A} \times \mathcal{I} \to [0, \tau]$, with each solver having a runtime limit of τ. The* New-Solver Problem *is about selecting benchmark instances from $\mathcal{I}$ to maximize the confidence in predicting the rank of a new solver $\hat{a} \notin \mathcal{A}$, while minimizing the summed runtime of $\hat{a}$ over all selected instances.*

The two objectives of this problem are conflicting, i.e., the ranking's confidence typically benefits from more instances while the runtime criterion encourages less or at least shorter evaluations. The crux is finding a good trade-off.

The approach presented in this paper meets several criteria for benchmarking that are considered desirable (cf. Table 1). Rather than outputting a binary classification, namely 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 its performance is to existing solvers. This helps to identify the solvers that have the most similar runtimes more quickly. In particular, our approach enables *ranking* the new solver among a set of existing solvers in one benchmarking run. We show that predicting the exact per-instance runtime of the solvers, which would be a more challenging task, is not necessary for ranking. Instead, we work with discretized runtime labels. We use instance features and known solver runtimes to predict these labels. Furthermore, we minimize the *runtime* required for our approach to arrive at its conclusion. Moreover, we select benchmark instances *non-randomly* and *incrementally*. In particular, we consider runtime information from experiments already done when choosing the next instance. 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 [24].

Experiments. We evaluated our approach using the SAT Competition 2022 Anniversary Track dataset [2], consisting of 5355 instances and runtimes of 28 solvers. Cross-validation was performed by treating each solver as the new solver once and learning to predict its PAR-2 rank. On average, our predictions achieve a ranking accuracy of approximately 92 % with only approximately 10 % of the runtime required

Table 1: Comparison of features of our benchmark-selection approach, the static benchmark-selection approach by Hoos et al. [16], the algorithm configuration system SMAC [18], and the active-learning approaches by Matricon et al. [24].

Feature	Hoos [16]	SMAC [18]	Matricon [24]	Our approach
Ranking/Scoring	✓	✗	(✓)	✓
Runtime Minimization	✗	✓	✓	✓
Incremental/Non-Random	✗	✗	✓	✓
Scalability	✓	✓	✗	✓

to evaluate these solvers on the complete set of instances. Our entire source code[1] and experimental data[2] are available on GitHub.

Outline. The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 describes our active learning approach. Section 4 introduces the experimental design. Section 5 evaluates the experimental results. Section 6 concludes.

Disclaimer. This article is an extended and refined version of our conference paper titled "Active Learning for SAT Solver Benchmarking" [10]. In particular, we significantly extended the analysis of runtime-prediction and runtime-discretization approaches (cf. Section 4.4), which determined our chosen solver runtime model for instance selection. Furthermore, we expanded the evaluation of which benchmark instances are selected by our approach (cf. Section 5.3).

2 Related Work

Benchmarking is of great interest in numerous research areas and represents an active field of research in its own right. Studies have shown that the compilation of benchmark instances poses several challenges. Using biased benchmarks can easily lead to fallacious interpretations [8]. Benchmarking also involves several interchangeable elements, including the performance measures used, how the measures are aggregated, and how missing values are handled. Questionable research practices could alter these elements a-posteriori to meet expectations, thereby skewing the results [30].

The following sections discuss related work from the areas of static benchmark selection (cf. Section 2.1), algorithm configuration (cf. Section 2.2), incremental benchmark selection (cf. Section 2.3), and active learning (cf. Section 2.4). Table 1 compares the most relevant approaches, which all pursue slightly different goals. In particular, our approach is *not* a general improvement over the others but the only one fully aligned with Definition 1.

2.1 Static Benchmark Selection

Benchmark selection is essential for solver competitions like the SAT Competition. In such competitions, the organizers define the rules for composing the benchmarks. These selection strategies are primarily static since they do not depend on particular

[1] https://github.com/mathefuchs/al-for-sat-solver-benchmarking
[2] https://github.com/mathefuchs/al-for-sat-solver-benchmarking-data

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 [9]. Manthey and Möhle propose an approach to remove redundancy from competition benchmarks by considering feature equivalence of formulas [23]. Mısır also presents a feature-based approach to reduce benchmarks by using matrix factorization and clustering [27].

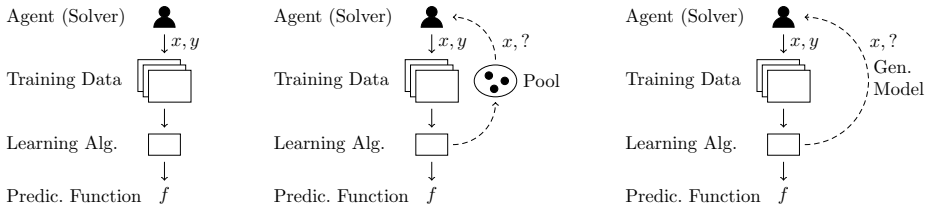
Hoos et al. [16] discuss desirable properties of SAT benchmark instances. They identify three key selection criteria: instance variety to avoid over-fitting, adapted instance hardness (neither too easy nor too hard), and avoiding duplicate instances. To filter instances that are too similar, they employ a distance-based approach with the SATzilla features [40, 41]. It should be noted, however, that the approach does not optimize for benchmark *runtime*. Instead, instances are selected *at random* except considering constraints on instance hardness and feature distance.

2.2 Algorithm Configuration

Further related work can be found within the field of algorithm configuration [17, 35], as exemplified by the configuration system SMAC [18]. Thereby, the goal is to tune SAT solvers for a given sub-domain of problem instances. While this task differs from our goal, for instance, in that we do not need to navigate the configuration space, there are similarities to our approach as well. For example, SMAC employs an iterative, model-based selection procedure similar to our approach, but this is for selecting configurations rather than instances. In contrast, instance selection is made *randomly* in SMAC, i.e., without building a model over instances. Furthermore, since algorithm configuration is designed to identify the optimal configuration, an algorithm configurator cannot be used to *rank* or *score* a new solver relative to others.

2.3 Incremental Benchmark Selection

Matricon et al. present an incremental benchmark selection approach [24]. Their *per-set efficient algorithm selection problem* (PSEAS) is similar to our *New-Solver Problem*, as given in Definition 1. To compare *a pair of* SAT solvers, the authors propose an iterative approach to select a solver-specific subset of instances until a desired confidence level is reached. This is achieved by calculating a scoring metric for all unselected instances, running the experiment with the highest score, and updating the confidence. Their approach ticks off most of the desired features in Table 1. However, the approach only compares solvers pairwise rather than providing a multi-solver *scoring* or *ranking*. Consequently, it is unclear how similar two given solvers are or on which instances they behave similarly. Furthermore, a significant shortcoming is the lack of *scalability* with the number of solvers. Due to comparing only pairs of solvers, evaluating a new solver with their method requires sampling an individual set of benchmark instances for each existing solver. In particular, comparing against one existing solver will typically require a different set of instances than comparing against another one. In contrast, our approach allows comparing a new solver against a set of existing solvers by sampling only one set of benchmark instances.



(a) Passive. (b) Active Pool-based. (c) Active Synthesis-based.

Fig. 1: Types of machine learning (depiction inspired by Rubens et al. [32]).

2.4 Active Learning

Our benchmark selection approach uses active learning. In passive machine learning, prediction models are trained on datasets with given instance labels (cf. Fig. 1a). In contrast, active learning (AL) begins with little or no labeled data. AL then repeatedly selects interesting problem instances for which to acquire labels, gradually improving the prediction model (cf. Fig. 1b). AL methods are particularly advantageous when acquiring labels is computationally expensive, such as obtaining solver runtimes. Without AL methods, it is not evident which instances to label and which not. In our scenario, the first goal is to maximize the utility of instances to our prediction model, which is measured by the ranking accuracy. Our second goal is to minimize the cost of label acquisition, which is estimated by the instance's predicted runtime. Consequently, our overall objective is to develop an accurate prediction model without labeling each data point.

Rubens et al. [32] survey active-learning advances. While synthesis-based AL methods [6, 11, 37] generate instances for labeling, pool-based methods [13, 15, 22] rely on a fixed set of unlabeled instances to sample from. Recent synthesis-based methods within the field of SAT solving demonstrate the generation of problem instances with desired properties [6, 11]. However, this objective is distinct from ours. While those approaches aim to generate instances on which a solver is either effective or ineffective, our objective is to predict whether a solver is effective or ineffective on an *existing* benchmark. In the latter direction, Volpato and Guangyan employ pool-based AL to train an instance-specific algorithm selector [38]. However, their objective is not to benchmark a solver's overall performance but to recommend the best solver for each SAT instance from a set of available solvers.

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 select instances repeatedly (Line 4) for benchmarking the new solver $\hat{a}$ (Line 5). The acquired result is subsequently employed to update the prediction model (Line 7). When the stopping criterion is met (Line 3), the benchmarking loop terminates, and the final score of $\hat{a}$ is

Algorithm 1: Incremental Benchmarking Framework

`Input:` Solvers $\mathcal{A}$, Instances $\mathcal{I}$, Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$, Solver $\hat{a}$
`Output:` Predicted Score of $\hat{a}$, Measured Runtimes $\mathcal{R}$

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

estimated (Line 8). Algorithm 1 returns this score, along with the acquired benchmark instances and runtime measurements.

Section 3.1 describes the underlying prediction model $\mathcal{M}$ and outlines the methodology for deriving a solver ranking from it. Section 3.2 presents the criteria for selecting instances. Finally, Section 3.3 presents potential stopping conditions.

3.1 Solver Model

The model $\mathcal{M}$ provides a prediction function $f_{\hat{a}} : \mathcal{I} \to \mathbb{R}$ for the new solver $\hat{a}$. This prediction function powers instance selection as described in Section 3.2. In the update step (Algorithm 1, Line 7), the model is trained to predict a transformed version of solver runtime (cf. Section 3.1.1) for the solver $\hat{a}$, using the features described in Section 4.2 and the previously acquired runtimes $\mathcal{R}$. Each iteration trains a new prediction model from scratch since the time for running the solver $\hat{a}$ on one instance e (Line 5) dominates the training time of the model $\mathcal{M}$ on all acquired instances $\mathcal{R}$ by a significant margin. Once the benchmarking loop terminates, the score of the solver $\hat{a}$ is predicted and then returned (Lines 8–9).

3.1.1 Runtime Transformation

For the prediction model $\mathcal{M}$, 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 $C_1, \ldots, C_k$ such that the fastest runtimes for the benchmark 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 $\gamma_k : \mathcal{A} \times \mathcal{I} \to \{1, \ldots, k\}$ is then specified as follows:

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

Empirical studies on portfolio solvers have demonstrated that discretization is an effective approach in practice [5, 29]. Section 4.4 presents a detailed analysis of the runtime transformation in our scenario, including the clustering approach. In particular, the analysis demonstrates that our benchmarking approach achieves higher ranking accuracy when working with discrete runtime labels rather than raw runtimes.

3.1.2 Ranking Solvers

To determine the rank of solvers, we apply PAR-2 scoring to the discrete runtime labels $\gamma_k(a, e)$ and obtain the adapted scoring function $s_k : \mathcal{A} \to [1, 2 \cdot k]$:

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

In this way, timeouts are still penalized.

3.2 Instance Selection

Selecting an instance based on the prediction 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 prediction function f and are inspired by existing work on active learning [33]. These methods require the model's predictions to include probabilities for the k discrete runtime labels. Let $f'_{\hat{a}} : \mathcal{I} \to [0, 1]^k$ 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. This is achieved by selecting the instance $e \in \mathcal{I} \setminus \tilde{\mathcal{I}}$ that minimizes the certainty $U(e)$, which is specified by the following equation:

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

Information-Gain Sampling

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

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

In the equation, $\mathrm{H}(e)$ denotes the entropy of the runtime labels $\gamma(a, e)$ over all $a \in \mathcal{A}$ and $\hat{\mathrm{H}}(e)$ denotes the entropy of these labels plus the runtime label for $\hat{a}$. The term

$\hat{\mathrm{H}}(e)_n$ is computed for every possible runtime label $n \in \{1, \ldots, k\}$. By maximizing the information gain, one favors instances whose most likely cluster labels, as determined by $f'_{\hat{a}}$, are similar to a great number of existing solvers. This decreases the entropy among the cluster labels of the given instance and we know that the new solver performs similarly to a number of solvers.

3.3 Stopping Criteria

In this section, we present the two dynamic stopping criteria (cf. Algorithm 1, Line 3) employed in our experiments: the Wilcoxon and the ranking stopping criterion. In the worst case, both criteria sample all instances, but typically they terminate earlier. How early they terminate depends on how easy it is to determine the rank of the new solver among the existing ones.

Wilcoxon Stopping Criterion

The Wilcoxon stopping criterion terminates the active-learning process based on the confidence that the predicted runtime labels of the new solver are sufficiently different from those of existing solvers. This criterion is loosely inspired by Matricon et al. [24]. To assess the statistical significance of the predicted runtime labels, the criterion uses the average p-value $W_{\hat{a}}$ of a Wilcoxon signed-rank test $\mathrm{w}(S, P)$ of the runtime label distributions $S = \{\gamma(a, e) \mid e \in \mathcal{I}\}$ for an existing solver a and $P = \{f_{\hat{a}}(e) \mid e \in \mathcal{I}\}$ of the new solver $\hat{a}$:

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

To improve the stability of this criterion and mitigate the impact of outliers, we employ an exponential moving average $W^{(i)}_{\mathrm{exp}}$:

$$W^{(0)}_{\mathrm{exp}} := 1$$
$$W^{(i)}_{\mathrm{exp}} := \beta W_{\hat{a}} + (1 - \beta)\, W^{(i-1)}_{\mathrm{exp}}$$

Active learning terminates when the value of $W^{(i)}_{\mathrm{exp}}$ drops below a fixed threshold.

Ranking Stopping Criterion

In contrast, the ranking stopping criterion is less sophisticated than the previous one. In this case, active learning terminates when the ranking induced by the model's predictions (cf. Equation (1)) has remained unchanged within the last l iterations. While the concrete values of the predicted score $s_{\hat{a}}$ may still change, the induced ranking is of sole interest in this criterion.

4 Experimental Design

This section outlines the experimental design employed in the presented study. This includes an overview of the evaluation framework (cf. Section 4.1), the datasets employed (cf. Section 4.2), the hyperparameter configuration (cf. Section 4.3), the runtime discretization (cf. Section 4.4), and the implementation (cf. Section 4.5).

Algorithm 2: Evaluation Framework

```
Input: Solvers 𝒜, Instances ℐ, Runtimes r : 𝒜 × ℐ → [0, τ]
Output: Average Ranking Accuracy Ō_acc, Average Fraction of Runtime Ō_rt
1  O ← ∅
2  for â ∈ 𝒜 do
3      𝒜' ← 𝒜 \ {â}
4      (s_â, ℛ) ← runALAlgorithm(𝒜', ℐ, r, â)        // Refer to Algorithm 1
       // Determine Ranking Accuracy
5      O_acc ← 0
6      for a ∈ 𝒜' do
7          if sgn(s_k(a) − s_â) = sgn(par_2(a) − par_2(â)) then   // same sign/order
8              O_acc ← O_acc + 1/|𝒜'|
       // Determine Runtime Fraction
9      t_total ← Σ_{e∈ℐ} r(â, e)
10     O_rt ← 0
11     for e ∈ ℐ do
12         if ∃t, (e, t) ∈ ℛ then
13             O_rt ← O_rt + t/t_total
14     O ← O ∪ {(O_acc, O_rt)}
15 (Ō_acc, Ō_rt) ← average(O)
16 return (Ō_acc, Ō_rt)
```

4.1 Evaluation Framework

To assess our active-learning framework, we perform cross-validation over a set of solvers (cf. Algorithm 2). Each solver assumes the role of the new solver $\hat{a}$ once (Line 2) and is therefore excluded from the set of solvers $\mathcal{A}$ in that iteration (Line 3). After running active learning for solver $\hat{a}$ (Line 4), we compute the value of two optimization goals, i.e., ranking accuracy and runtime. The *ranking accuracy* $O_{\text{acc}} \in [0, 1]$ (higher is better) is defined as the fraction of correctly ranked solver pairs $(\hat{a}, a)$ regarding the PAR-2 scoring, where a belongs to the set of all solvers $\mathcal{A}$ (Lines 5–8). The *fraction of runtime* required by the framework to reach its conclusion is represented by $O_{\text{rt}} \in [0, 1]$ (lower is better). This metric compares the accumulated runtimes over the sampled instances to the accumulated runtimes over all instances in the dataset (Lines 9–13). Once all cross-validation results have been collected (Line 14), the output metrics are averaged and returned (Lines 15–16).

To compare different instantiations of our active-learning framework holistically, we insert the cross-validation results into Equation (2):

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

$\delta \in [0, 1]$ is a factor for weighting the two optimization goals O_{acc} and O_{rt}. Plotting the approaches that maximize O_δ for all $\delta \in [0, 1]$ on an O_{rt}-O_{acc}-diagram provides us with a Pareto front of the best approaches for different optimization-goal weightings.

4.2 Data

In our experiments, we work with the dataset of the SAT Competition 2022 Anniversary Track [2, 20]. This dataset consists of 5355 instances with corresponding runtimes of 28 sequential SAT solvers. For predictions within our framework, we also use a database of 56 instance features from the Global Benchmark Database (GBD) [19]. Refer to Appendix A for a full list of all 56 features. They include instance size features and node distribution statistics for several graph representations of SAT instances, and are primarily inspired by the SATzilla 2012 features described in [41]. All features are numeric and have no missing values. We drop 10 of the 56 features due to zero variance. In total, the prediction models have access to 46 instance features and 27 out of 28 runtime features, excluding the respective new solver $\hat{a}$.

In addition, we retrieve instance family information[3] to evaluate the composition of our sampled benchmarks. Instance families consist of instances from the same application domain, such as planning, cryptography, etc., and are valuable for analyzing solver performance. For hyperparameter tuning, we randomly sample 10 % of the complete set of 5355 instances with stratification regarding the instance family. All instance families that are too *small*, i.e., families with less than 10 instances, are put into one meta-family for stratification. This *tuning dataset* allows a more extensive exploration of the hyperparameter space.

4.3 Hyperparameters

There are several possible instantiations for the three subroutines *ranking*, *selection*, and *stopping* in Algorithm 1. We describe these experimental configurations next.

Ranking. Regarding *ranking* (cf. Section 3.1), we experiment with the following instantiations:

- Observed PAR-2 ranking of already sampled instances, without using prediction
- Predicted runtime-label ranking
  - History size: For stability, consider the latest 1, 10, 20, 30, or 40 predictions within a voting approach. The latest x predictions for each instance vote on the instance's winning label.
  - Fallback threshold: If the difference of scores between the new solver $\hat{a}$ and another solver drops below 0.01 , 0.05 , or 0.1 , use the partially observed PAR-2 ranking as a tie-breaker.

Selection. We experiment with the following instantiations of the instance *selection* (cf. Section 3.2). Since the potential runtime of the solver experiments is orders of magnitude larger than the model update time, we increment our benchmark one instance at a time rather than using batches of multiple instances. Both these strategies are

[3] https://benchmark-database.de/getdatabase/meta

used in the state-of-the-art in active learning [34, 37]. A drawback of selecting instances one at a time is the lack of parallel execution of runtime experiments.

- Random sampling
- Uncertainty sampling
  - Fallback threshold: Use random sampling for the first 0 %, 5 %, 10 %, 15 %, or 20 % of instances to explore the instance space.
  - Runtime scaling: Whether to scale the uncertainty scores by the average runtime of solvers per instance or use $\mathrm{U}(e)$ as is.
- Information-gain sampling
  - Fallback threshold: Use random sampling for the first 0 %, 5 %, 10 %, 15 %, or 20 % of instances to explore the instance space.
  - Runtime scaling: Whether to scale the information-gain scores by the average runtime of solvers per instance or use $\mathrm{IG}(e)$ as is.

Stopping. We evaluate the following *stopping* (cf. Section 3.3) criteria:

- Subset-size stopping criterion: Stop after sampling 10 % or 20 % of instances.
- Ranking stopping criterion
  - Minimum amount: Sample at least 2 %, 8 %, 10 %, or 12 % of instances before applying the criterion.
  - Convergence duration: Stop if the predicted ranking stays the same for a number of sampled instances equal to 1 % or 2 % of all instances.
- Wilcoxon stopping criterion
  - Minimum amount: Sample at least 2 %, 8 %, 10 %, or 12 % of instances before applying the criterion.
  - Average of p-values to drop below: 5 %.
  - Exponential moving average: Incorporate previous significance values by using an EMA with $\beta = 0.1$ or $\beta = 0.7$.

4.4 Runtime Discretization and Prediction

One of the most important parts of our framework is the solver runtime model (cf. Section 3.1). In this section, we detail how clustering solver runtimes can lead to informative runtime labels and which machine learning models are best at predicting them. To this end, we analyze prediction performance for estimating runtimes, timeouts, and discretized runtimes.

Regressing Runtime

As a baseline, Table 2 shows the prediction performance of two regression models, i.e., a random forest and a multilayer perceptron. We employ these models to estimate a new solver's raw runtime as the prediction target. To this end, we use 46 instance features (Section 4.2) and 27 known solver runtimes as inputs for the predictions. We repeat this evaluation for each solver as the prediction target and report the average and standard deviation of prediction performance over the 28 solvers.

Table 2: Test-set prediction performance for predicting solver runtime (a continuous target) with two prediction models.

Regression Models	Avg. RMSE (± std)	Avg. MAE (± std)
Random Forest	1929.84 s (± 1161.15 s)	957.36 s (± 297.21 s)
Multilayer Perceptron	2275.20 s (± 1545.01 s)	1284.30 s (± 390.57 s)

Table 3: Test-set prediction performance for predicting solver timeouts (a binary target) with different prediction models.

Timeout Prediction Models	Average MCC (± std)	
	Without Runtime Feat.	With Runtime Feat.
Stacking (QDA + RF)	0.6513 (± 0.0458)	0.9527 (± 0.0292)
Quadratic Discriminant Analysis (QDA)	0.2593 (± 0.1310)	0.9290 (± 0.0339)
Random Forest (RF)	0.6607 (± 0.0547)	0.8530 (± 0.0479)
AdaBoost	0.5412 (± 0.0612)	0.8384 (± 0.0444)
Decision Tree	0.5980 (± 0.0423)	0.8059 (± 0.0707)
Logistic Regression	0.2031 (± 0.0728)	0.8052 (± 0.1018)
kNN	0.5108 (± 0.0387)	0.7885 (± 0.1521)
Multilayer Perceptron	0.1293 (± 0.0718)	0.7760 (± 0.1408)
Support Vector Machine	0.0595 (± 0.0554)	0.7757 (± 0.2149)
Naive Bayes	0.1173 (± 0.0872)	0.7306 (± 0.1394)

We chose both types of prediction models for their robustness and ability to capture nonlinear dependencies [3]. However, both still perform poorly, as evidenced by the magnitudes of the mean regression errors in Table 2. In particular, a root mean squared error (RMSE) of about 2000 s is relatively high for solver runtimes that range from 0 to 5000 s. Such unreliable runtime estimates could affect active learning negatively. Therefore, we decided to discretize the runtimes for prediction, moving from a regression scenario to a classification scenario.

Classifying Timeouts

Table 3 shows the prediction performance for estimating whether a solver times out or not. This binary prediction target represents a very strong discretization of runtimes. We compare a variety of prediction models that pursue different learning paradigms. Each is trained with either instance features only (second column) or instance and runtime features (third column). As the evaluation metric for prediction performance, we use Matthews Correlation Coefficient (MCC) [14, 25], which reaches its maximum of 1 for a perfect prediction, is 0 for random guessing, and has a minimum of -1.

Training the classification models with instance features only, a random forest performs best. In contrast, a quadratic discriminant analysis performs poorly in this scenario but performs very well if runtime features are included. However, the best prediction performance with all features is achieved by a stacking ensemble [39] out of the quadratic discriminant analysis [36] and the random forest [3]. Stacking means

Table 4: Test-set prediction performance for predicting the C_k cluster labels (a discrete target) with the best-performing models.

C_3 Cluster Prediction Models	Average MCC (± std)	
	Without Clustering Feat.	With Clustering Feat.
Stacking (QDA + RF)	0.7464 (± 0.0497)	0.8380 (± 0.0634)
Quadratic Discriminant Analysis (QDA)	0.6903 (± 0.0607)	0.7738 (± 0.0459)
Random Forest (RF)	0.7116 (± 0.0385)	0.7841 (± 0.0469)
C_4 Cluster Prediction Models	Average MCC (± std)	
	Without Clustering Feat.	With Clustering Feat.
Stacking (QDA + RF)	0.6562 (± 0.0538)	0.7366 (± 0.0377)
Quadratic Discriminant Analysis (QDA)	0.6007 (± 0.0765)	0.6872 (± 0.0469)
Random Forest (RF)	0.6033 (± 0.0455)	0.6828 (± 0.0527)
C_5 Cluster Prediction Models	Average MCC (± std)	
	Without Clustering Feat.	With Clustering Feat.
Stacking (QDA + RF)	0.5835 (± 0.0615)	0.6396 (± 0.0553)
Quadratic Discriminant Analysis (QDA)	0.5508 (± 0.0669)	0.5881 (± 0.0912)
Random Forest (RF)	0.5420 (± 0.0459)	0.5895 (± 0.0475)

that another model, in our case a simple decision tree [4], acts as a meta-model: The prediction outputs of the ensemble members, i.e., quadratic discriminant analysis and random forests, become the prediction inputs of the meta-model, i.e., decision tree.

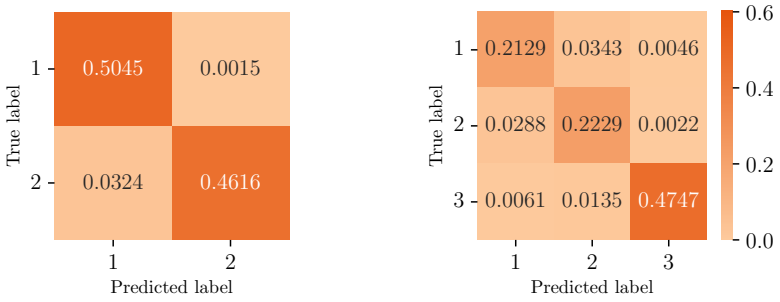
Classifying Discretized Runtime Labels

Building on the success of binary timeout prediction, we investigate a classification scenario where the target is solver runtime discretized into multiple categories.

Label definition. To define this prediction target, we partition the solver runtimes into k clusters. We conduct this clustering step per instance, i.e., the solvers forming a particular cluster may differ from instance to instance. The adapted solver scoring function s_k (cf. Equation (1)) subsequently uses the cluster labels. In preliminary experiments, the most *useful* labels were produced with hierarchical clustering and a log-single-linkage criterion; we will discuss our notion of label usefulness later. Other clustering approaches we have tried include hierarchical clustering with mean-, median-, and complete-linkage criteria, as well as k-means and spectral clustering.

In our chosen hierarchical clustering procedure, each non-timeout runtime starts in a separate cluster. We then gradually merge the closest clusters until the desired number of $k-1$ partitions is reached. By definition, the k-th cluster always contains the timeouts and no other runtimes (cf. Section 3.1.1). To quantify the distance between two clusters, we consider the minimum pairwise difference of logarithmic runtimes between solvers from the two clusters.

Label prediction. Next, we use the runtimes labels from this discretization procedure as prediction targets. Table 4 shows the prediction performance for the best classification models from Table 3 and $k \in \{3, 4, 5\}$ clusters. We train the prediction models



(a) Timeout prediction.

(b) C_3 cluster prediction.

Fig. 2: Confusion matrices for predicting timeouts and C_3 cluster labels.

either with 46 instance features (Section 4.2) and 27 known solver runtimes (second column) or additionally including 27 features representing the known C_k cluster labels of other solvers (third column).

For all prediction models and values of k, the prediction performance is better when the cluster labels are included as features. In addition, the prediction performance degrades rapidly as the number of clusters increases. The stacking ensemble performs best in all cases, so we use it in all subsequent experiments.

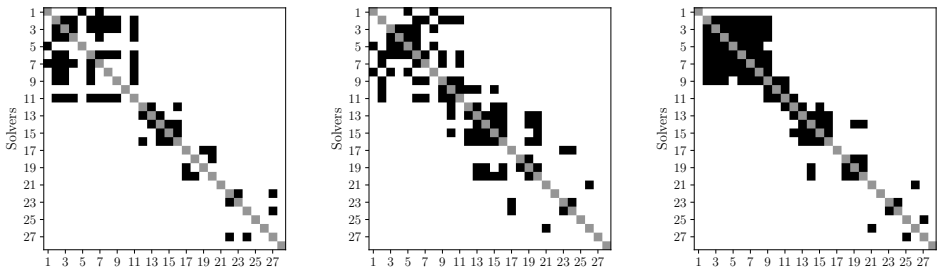
To analyze the stacking ensemble in more depth, Fig. 2 shows the confusion matrices for predicting timeouts and C_3 cluster labels. Both scenarios admit relatively good prediction performance, with the model for C_3 performing slightly worse than the model for timeouts. Notably, for C_3, there is less confusion between the non-timeout labels (Labels 1 and 2) and the timeout label (Label 3) than between the non-timeout labels (Labels 1 and 2). This observation suggests that deciding on timeouts is easier than distinguishing between discretized non-timeout runtimes.

Label usefulness. To ensure that the runtime labels also are *useful* for active learning, we evaluate whether the labels still discriminate solvers and agree with the actual PAR-2 ranking. Table 5 shows the ranking of the solvers in the SAT Competition 2022 Anniversary Track [2] according to the standard PAR-2 score and according to our discretized s_3 score. We observe that ranking with our score correctly decides for almost all (about 97.45 %; $\sigma = 3.68$ %) solver pairs which solver is faster. In particular, the Spearman correlation of s_3 and the PAR-2 score over all solvers is about 0.988, which is very close to the optimal value of 1 [7].

We also analyze how runtime discretization affects the distinguishability of solver pairs. The matrices in Fig. 3 visualize which solver pairs exhibit significant differences in their PAR-2 scores, timeouts, and s_3 scores. According to a Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$, 87.57 % of the solver pairs have significantly different scores after γ_3-discretization, which is only a slight decrease compared to 90.48 % before discretization. γ_2-discretization significantly distinguishes 87.30 % of solver pairs. These results support the usefulness of discretized runtimes for our active learning framework.

Table 5: Ranking of solvers for the PAR-2 score and our discretized s_3 score.

PAR-2		s_3		Solver
Rank	Score	Rank	Score	
1	2808.13	1	1.1717	kissat-mab-esa
2	2812.93	2	1.1832	kissat-sc2022-bulky
3	2835.25	3	1.1862	ekissat-mab-gb-db
4	2835.59	4	1.1868	kissat-mab-ucb
5	2836.92	5	1.1868	kissat-inc
6	2845.19	6	1.1926	ekissat-mab-db-v1
7	2846.73	7	1.1930	kissat-mab-moss
8	2857.67	8	1.1947	kissat-mab-hywalk
9	2869.45	9	1.1998	kissat-sc2022-light
10	2899.70	10	1.2164	kissat-els-v2
11	2953.59	11	1.2290	hkis-unsat
12	2967.53	12	1.2347	kissat-adaptive-restart
13	2976.56	13	1.2475	seqfrost-noextend
14	3014.40	16	1.2645	kissat-els-v1
15	3017.73	14	1.2509	cadical-esa
16	3036.83	15	1.2613	cadical-reorder
17	3049.90	20	1.3648	cadical-rel-scavel
18	3080.66	19	1.2965	kissat-relaxed
19	3095.73	17	1.2815	cadical-dvdl-v1
20	3101.12	18	1.2856	cadical-dvdl-v2
21	3273.95	21	1.3786	glucose-reboot
22	3290.90	25	1.4707	lstech-maple-hywalk
23	3292.68	23	1.4478	lstech-maple
24	3400.72	24	1.4693	slime-sc-2022-beta
25	3412.11	26	1.5237	slime-sc-2022
26	3436.07	22	1.4161	hcad-v1-psids
27	3506.32	27	1.5433	maple-lcmdistchrbt-dl-v3
28	4741.50	28	2.0581	isasat



(a) PAR-2 scores. (b) Timeouts. (c) s_3 scores.

Fig. 3: Solver pairs with significant differences regarding their scorings are colored white, non-significant ones black. Solvers are ordered identically to Table 5.

4.5 Implementation Details

For reproducibility, our source code and data are available on GitHub (see footnotes in Section 1). Our code is implemented in PYTHON using *scikit-learn* [31] for making predictions, and *gbd-tools* [19] for retrieving SAT instances.

5 Evaluation

In this section, we evaluate our active learning framework. First, we analyze and tune the different subroutines of our framework on the tuning dataset (cf. Section 5.1). Next, we evaluate the best configurations on the full dataset (cf. Section 5.2). Finally, we analyze the importance of instance families for our framework (cf. Section 5.3).

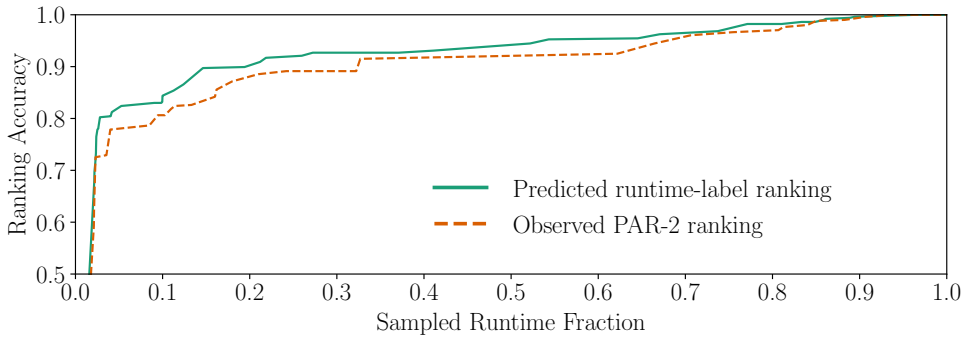
5.1 Hyperparameter Analysis

Methodology. Fig. 4 shows the performance of the hyperparameter configurations from Section 4.3 on O_{rt}-O_{acc} plots (cf. Section 4.1) for the hyperparameter-tuning dataset. Evaluating one configuration with Algorithm 2 returns one point (O_{rt}, O_{acc}). We do not show intermediate results of the active learning procedure, but only the final results after stopping. The plotted lines represent the best-performing configurations per ranking approach (Fig. 4a), selection approach (Fig. 4b), and stopping criterion (Fig. 4c). In particular, we show the Pareto front, i.e., from all configurations that share a particular value of the plotted hyperparameter, we take the maximum ranking accuracy over all remaining hyperparameters *not* shown in the corresponding plot.

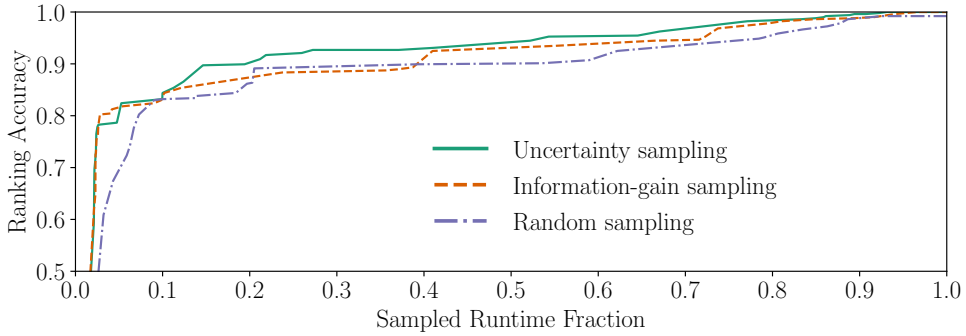
Ranking. Regarding ranking approaches (Fig. 4a), the predicted runtime-label ranking outperforms the partially observed PAR-2 ranking consistently for every possible value of the tradeoff parameter δ. This result is expected since selection decisions are usually not random. For example, we might sample more instances of a particular family if it benefits the discrimination of solvers. While such a sampling biases the partially observed PAR-2 score, the prediction model can account for this.

Selection. Regarding the selection approaches (Fig. 4b), uncertainty sampling performs best in most cases. However, information-gain sampling is advantageous when runtime is strongly favored (small δ; runtime fraction less than 5 %). This result is consistent with our expectations: Information-gain sampling selects instances that maximize the expected reduction in entropy. This means that we sample instances that reveal similarities rather than differences between solvers, which helps to build a confident model quickly. However, the method cannot select instances that are useful for distinguishing solvers later. Random sampling performs reasonably well but is outperformed by uncertainty sampling in all cases, demonstrating the benefit of actively selecting instances based on a prediction model.

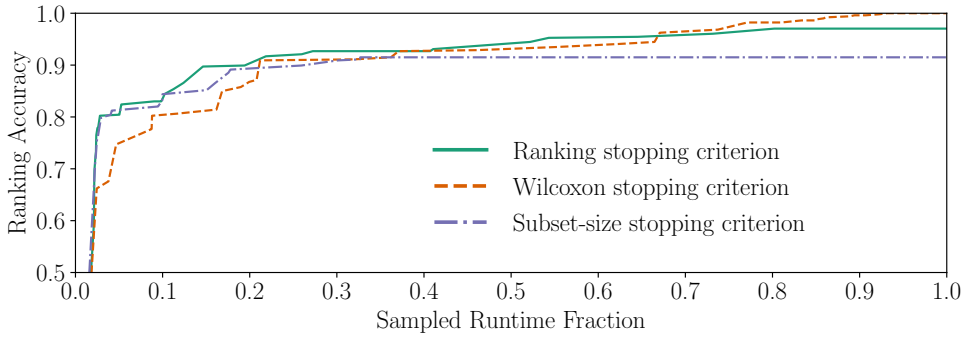
Stopping. Regarding the stopping criteria (Fig. 4c), the ranking stopping criterion performs most consistently well. When accuracy is strongly favored (very high δ), the Wilcoxon stopping criterion performs better. The subset-size stopping criterion performs reasonably well, but does not improve beyond a certain accuracy because it samples a fixed subset of instances, independent of prediction performance.



(a) Ranking approaches.

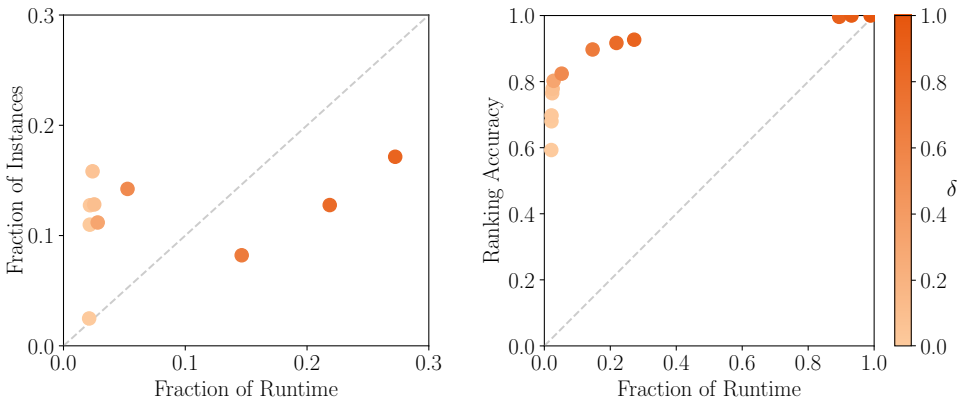


(b) Selection approaches.



(c) Stopping criteria.

Fig. 4: O_{rt}-O_{acc}-diagrams (cf. Section 4.1) comparing different hyperparameter instantiations of our active-learning framework on the hyperparameter-tuning dataset. The x-axis shows the ratio of total solver runtime on the sampled instances relative to all instances. The y-axis shows the ranking accuracy. Each line represents the front of Pareto-optimal configurations for the respective hyperparameter instantiation.



(a) Runtime vs. Instances

(b) Runtime vs. Accuracy

Fig. 5: Scatter plot comparing different instantiations of the tradeoff parameter δ for our active learning framework on the hyperparameter-tuning dataset. The x-axes show the fraction of runtime O_{rt} of the sample, while the y-axes show the fraction of sampled instances and the ranking accuracy, respectively. The color indicates the weighting between different optimization objectives $\delta \in [0, 1]$. The larger δ, the more we favor accuracy over runtime.

Objective weighting. Fig. 5a shows an interesting consequence of weighting our optimization goals: On the one hand, if we want to get a *rough* estimate of a solver's performance quickly (low δ), our framework favors selecting many *easy* instances. In particular, the fraction of sampled instances is larger than the fraction of runtime. By having many observations, it is easier to build a model. On the other hand, if we want to get a *good* estimate of a solver's performance in a moderate amount of time (high δ), our framework favors selecting few, *difficult* instances. In particular, the fraction of instances is smaller than the fraction of runtime. Furthermore, Fig. 5b shows which values of δ make the most sense. The range $\delta \in [0.2, 0.8]$ corresponds to the points with a runtime fraction between 0.03 and 0.22 We consider this region to be the most promising, analogous to the *elbow* method in cluster analysis [21].

5.2 Full-Dataset Evaluation

After selecting the most promising hyperparameters, we run our active learning experiments on the complete Anniversary Track dataset (5355 instances). The above range $\delta \in [0.2, 0.8]$ results in only two different hyperparameter configurations. The best-performing approach for $\delta \in [0.2, 0.7]$ uses the predicted runtime label ranking, information-gain sampling, and ranking stopping criterion. It can predict the PAR-2 ranking of a new solver with 90.48 % accuracy (O_{acc}) in only 5.41 % of the total evaluation time (O_{rt}). The best-performing approach for $\delta \in (0.7, 0.8]$ uses the predicted runtime label ranking, uncertainty sampling, and ranking stopping criterion. It

Table 6: Performance comparison (on the full dataset) of the best-performing AL approach (*AL*), random sampling of the same runtime fraction with 1000 repetitions (*Random*), and static selection of the instances most frequently sampled by active learning approaches (*Most Frequent*).

For $\delta \in [0.2, 0.7]$	AL	Random	Most Frequent
Sampled Runtime Fraction (%)	5.41	5.43	5.44
Sampled Instance Fraction (%)	26.53	5.43	27.75
Ranking Accuracy (%)	90.48	88.54	81.08
For $\delta \in (0.7, 0.8]$	AL	Random	Most Frequent
Sampled Runtime Fraction (%)	10.35	10.37	10.37
Sampled Instance Fraction (%)	5.24	10.37	36.96
Ranking Accuracy (%)	92.33	91.61	84.52

can predict the PAR-2 ranking of a new solver with 92.33 % accuracy (O_{acc}) in only 10.35 % of the total evaluation time (O_{rt}).

Table 6 shows how these two active learning approaches (column *AL*) compare to two static baselines: *Random* samples instances until it reaches roughly the same fraction of runtime as the AL benchmark sets. We repeat the sampling 1000 times and report the average results. *Most Frequent* uses a static benchmark set consisting of the instances most frequently sampled by our active learning approach. In particular, we consider the average sampling frequency across all solvers and Pareto-optimal active learning approaches.

Both of our AL approaches outperform random sampling. However, the performance differences are not significant for a Wilcoxon signed-rank test with $\alpha = 0.05$ and also depend on the fraction of the runtime sampled (cf. Fig. 4b). However, a clear advantage of our approach is that it indicates when to stop adding new instances. While the active learning results on the full dataset are not as strong as on the smaller tuning dataset, they still show the benefit of making the benchmark selection dependent on the solvers to be distinguished.

A static benchmark utilizing the most frequently AL-sampled instances exhibits suboptimal performance compared to active learning and random sampling. This outcome is anticipated since the static benchmark fails to reflect an appropriate balance of instance families.

5.3 Instance Frequency and Instance-Family Importance

Fig. 6 illustrates the frequency distribution with which individual instances are selected across all active learning benchmarks regarding the full dataset with cross-validation over all solvers. A small proportion of instances is selected with great frequency, while most instances are selected on only a few occasions. This indicates that there is no fixed subset of instances that can perfectly distinguish all solvers, which highlights the value of employing an active-learning-based strategy.

The selection decisions of our approach also reveal the importance of instance families for our framework. Fig. 7 illustrates the occurrence of instance families within the dataset and the benchmarks created by active learning. We use the best-performing

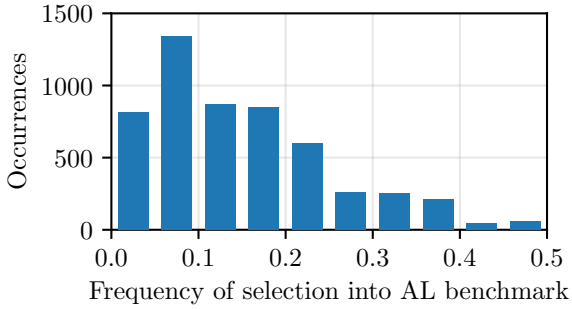


Fig. 6: Frequency with which individual problem instances are selected within all active-learning benchmarks. Instances that are chosen more frequently are intuitively more important for distinguishing solver runtimes. Results are on the full dataset with cross-validation over all solvers.

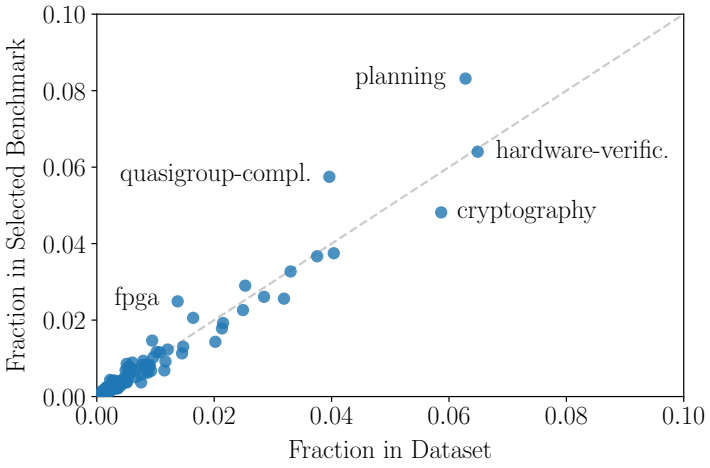


Fig. 7: Scatter plot illustrating the relative importance of different instance families for ranking on the full dataset. The x-axis depicts the frequency of instance families in the dataset, while the y-axis represents the average frequency in the samples selected by active learning. The dashed line represents families that occur with the same frequency in the dataset and samples. Results are on the full dataset with cross-validation over all solvers.

configurations for all values of $\delta \in [0, 1]$. While most families exhibit the same fraction in the dataset and the sampled benchmarks, a few outliers require further discussion. Instances of the families *fpga*, *quasigroup-completion*, and *planning* are particularly useful for distinguishing solvers within our framework. These instances are selected in greater proportion than in the full dataset. In contrast, instances of the largest family, i.e., *hardware-verification*, appear with approximately the same fraction in the dataset

and the sampled benchmarks. Finally, instances of the family *cryptography* are less important in distinguishing solvers than their substantial representation in the dataset would suggest. One possible explanation for this discrepancy is that these instances are highly similar, such that a small fraction of them is sufficient to estimate a solver's performance on all of them.

6 Conclusions and Future Work

In this work, we have addressed the *New-Solver Problem*: Given a new solver, we aim to determine its ranking relative to competitors. Our approach provides accurate ranking predictions while requiring significantly less runtime than a complete evaluation on a given benchmark set. On data from the SAT Competition 2022 Anniversary Track, we can determine a new solver's PAR-2 ranking with about 92 % accuracy while only requiring 10 % of the full-evaluation time. We have evaluated several ranking methods, instance-selection approaches, and stopping criteria within our sequential active learning framework. We have also looked at which instance families are the most prevalent in selection decisions.

Future work could compare more sub-routines for ranking, instance selection, and stopping. Further, our evaluation framework can be used for other computation-intensive problems. In particular, many $\mathcal{NP}$-complete problems share most of the relevant properties of SAT, like established instance features, a complete benchmark is expensive, and traditional benchmark selection requires expert knowledge.

From a technical perspective, we could formulate the problem of runtime discretization as an optimization problem rather than addressing it empirically. Further, a major shortcoming of our current approach is the lack of parallelization, selecting instances one at a time. Benchmarking on a computing cluster with n cores benefits from having batches of n instances. However, bigger batch sizes n impede *active learning*. Also, it needs to be clarified how to synchronize instance selection and updates of the prediction model without wasting too much runtime.

Declarations

Funding. This work was supported by the Ministry of Science, Research and the Arts Baden-Württemberg, project *Algorithm Engineering for the Scalability Challenge (AESC)*.

Competing interests. The authors have no competing interests to declare that are relevant to the content of this article.

Availability of data and materials. All experimental data are available online at https://github.com/mathefuchs/al-for-sat-solver-benchmarking-data.

Code availability. The code is available online at https://github.com/mathefuchs/al-for-sat-solver-benchmarking.

References

[1] Balint A, Belov A, Järvisalo M, et al (2015) Overview and analysis of the SAT Challenge 2012 solver competition. Artificial Intelligence 223:120–155

[2] Balyo T, Heule M, Iser M, et al (eds) (2022) Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions, Department of Computer Science, University of Helsinki, URL http://hdl.handle.net/10138/347211

[3] Breiman L (2001) Random forests. Machine Learning 45(1):5–32

[4] Breiman L, Friedman JH, Olshen RA, et al (1984) Classification and Regression Trees, 1st edn. Wadsworth

[5] Collautti M, Malitsky Y, Mehta D, et al (2013) SNNAP: solver-based nearest neighbor for algorithm portfolios. In: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, pp 435–450

[6] Dang N, Akgün Ö, Espasa J, et al (2022) A framework for generating informative benchmark instances. In: International Conference on Principles and Practice of Constraint Programming, pp 18:1–18:18

[7] De Winter JCF, Gosling SD, Potter J (2016) Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. Psychological Methods 21(3):273–290

[8] Dehghani M, Tay Y, Gritsenko AA, et al (2021) The benchmark lottery. CoRR abs/2107.07002

[9] Froleyks N, Heule M, Iser M, et al (2021) SAT Competition 2020. Artificial Intelligence 301

[10] Fuchs T, Bach J, Iser M (2023) Active learning for SAT solver benchmarking. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp 407–425

[11] Garzón I, Mesejo P, Giráldez-Cru J (2022) On the performance of deep generative models of realistic SAT instances. In: International Conference on Theory and Applications of Satisfiability Testing, pp 3:1–3:19

[12] Gelder AV (2011) Careful ranking of multiple solvers with timeouts and ties. In: International Conference on Theory and Applications of Satisfiability Testing, pp 317–328

[13] Golbandi N, Koren Y, Lempel R (2011) Adaptive bootstrapping of recommender systems using decision trees. In: International Conference on Web Search and Data Mining, pp 595–604

[14] Gorodkin J (2004) Comparing two k-category assignments by a k-category correlation coefficient. Computational Biology and Chemistry 28(5–6):367–374

[15] Harpale A, Yang Y (2008) Personalized active learning for collaborative filtering. In: Conference on Research and Development in Information Retrieval, pp 91–98

[16] Hoos HH, Kaufmann B, Schaub T, et al (2013) Robust benchmark set selection for boolean constraint solvers. In: Learning and Intelligent Optimization Conference, pp 138–152

[17] Hoos HH, Hutter F, Leyton-Brown K (2021) Automated configuration and selection of SAT solvers. In: Handbook of Satisfiability, 2nd edn. IOS Press, chap 12, p 481–507

[18] Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. In: Learning and Intelligent Optimization Conference, pp 507–523

[19] Iser M, Jabs C (2024) Global benchmark database. In: International Conference on Theory and Applications of Satisfiability Testing, pp 18:1–18:10

[20] Iser M, Heule M, Järvisalo M, et al (2024) Benchmarks of SAT Competition 2022: Anniversary track. https://doi.org/10.5281/zenodo.11175170

[21] Kodinariya TM, Makwana PR (2013) Review on determining number of cluster in k-means clustering. International Journal of Advance Research in Computer Science and Management Studies 1(6):90–95

[22] Koren Y, Bell RM, Volinsky C (2009) Matrix factorization techniques for recommender systems. Computer 42(8):30–37

[23] Manthey N, Möhle S (2016) Better evaluations by analyzing benchmark structure. In: Pragmatics of SAT

[24] Matricon T, Anastacio M, Fijalkow N, et al (2021) Statistical comparison of algorithm performance through instance selection. In: International Conference on Principles and Practice of Constraint Programming, pp 43:1–43:21

[25] Matthews BW (1975) Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochimica et Biophysica Acta 405(2):442–451

[26] Mısır M (2017) Data sampling through collaborative filtering for algorithm selection. In: IEEE Congress on Evolutionary Computation, pp 2494–2501

[27] Mısır M (2021) Benchmark set reduction for cheap empirical algorithmic studies. In: IEEE Congress on Evolutionary Computation, pp 871–877

[28] Mısır M, Sebag M (2017) ALORS: An algorithm recommender system. Artificial Intelligence 244:291–314

[29] Ngoko Y, Cérin C, Trystram D (2019) Solving SAT in a distributed cloud: A portfolio approach. International Journal of Applied and Computational Mathematics 29(2):261–274

[30] Nießl C, Herrmann M, Wiedemann C, et al (2022) Over-optimism in benchmark studies and the multiplicity of design and analysis options when interpreting their results. WIREs Data Mining and Knowledge Discovery 12(2)

[31] Pedregosa F, Varoquaux G, Gramfort A, et al (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12(85):2825–2830

[32] Rubens N, Elahi M, Sugiyama M, et al (2015) Active learning in recommender systems. In: Recommender Systems Handbook, 2nd edn. Springer, chap 24, p 809–846

[33] Settles B (2009) Active learning literature survey. Tech. rep., University of Wisconsin-Madison, Department of Computer Sciences, URL http://digital.library.wisc.edu/1793/60660

[34] Sinha S, Ebrahimi S, Darrell T (2019) Variational adversarial active learning. In: International Conference on Computer Vision, pp 5971–5980

[35] Stützle T, López-Ibáñez M, Pérez-Cáceres L (2022) Automated algorithm configuration and design. In: Genetic and Evolutionary Computation Conference, pp 997–1019

[36] Tharwat A (2016) Linear vs. quadratic discriminant analysis classifier: A tutorial. International Journal of Applied Pattern Recognition 3(2):145–180

[37] Tran T, Do T, Reid ID, et al (2019) Bayesian generative active deep learning. In: International Conference on Machine Learning, pp 6295–6304

[38] Volpato R, Song G (2019) Active learning to optimise time-expensive algorithm selection. CoRR abs/1708.07747

[39] Wolpert DH (1992) Stacked generalization. Neural Networks 5(2):241–259

[40] Xu L, Hutter F, Hoos HH, et al (2008) SATzilla: Portfolio-based algorithm selection for SAT. Journal Of Artificial Intelligence Research 32:565–606

[41] Xu L, Hutter F, Hoos HH, et al (2012) Features for SAT. Tech. rep., University of British Columbia, URL https://www.cs.ubc.ca/labs/beta/Projects/SATzilla/Report_SAT_features.pdf

A Instance Features

For predictions within our framework, we use a database of 56 instance features[4] from the Global Benchmark Database (GBD) [19]. Table 7 briefly summarizes all 56 features. The placeholder `<n>` can take values within the range 1 to 9 and the placeholder `<dist>` is one of `mean`, `variance`, `min`, `max`, or `entropy`. Accounting for all possible instantiations of `<n>` and `<dist>`, there are 56 features in total.

Table 7: List of all 56 instance features, where `<dist>` is one of `mean`, `variance`, `min`, `max`, or `entropy`.

Category	Feature Name	Description
Basic statistics	`clauses`	Number of clauses
	`variables`	Number of variables
Clause types	`cls<n>`	Number of clauses with `<n>` literals for `<n>` in 1–9
	`cls10p`	Number of clauses with 10 or more literals
	`horn`	Number of Horn clauses
	`invhorn`	Number of inverted Horn clauses
	`positive`	Number of positive clauses
	`negative`	Number of negative clauses
Horn proximity	`hornvars_<dist>`	Distribution of variable occurrence in horn clauses
	`invhornvars_<dist>`	Distribution of variable occurrence in inverted horn clauses
Polarity balance	`balancecls_<dist>`	Distributions of fractions of number of positive/negative literals in clauses
	`balancevars_<dist>`	Distributions of fractions of number of positive/negative literals of variables
Variable-clause graph	`vcg_vdegree_<dist>`	Variable degree distribution
	`vcg_cdegree_<dist>`	Clause degree distribution
Variable graph	`vg_degree_<dist>`	Degree distribution
Clause graph	`cg_degree_<dist>`	Degree distribution

[4] https://udopia.github.io/gbdc/doc/extractors/CNFBaseFeatures.html