Active Learning for SAT Solver Benchmarking

Tobias Fuchs, Jakob Bach and Markus Iser

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 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 [2, 8]. 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 [11, 14]. 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 $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$ 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 [22, 24].

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 code[1] and experimental data[2] are available on GitHub.

This article is an extended version of a conference paper with the same name [9]. In particular, we significantly extended the analysis of runtime-prediction approaches (cf. Section 4.4) and added an evaluation of how often problem instances were selected in benchmarks (cf. Section **??**).

[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. [14], the algorithm configuration system SMAC [16], and the active-learning approaches by Matricon et al. [21].

Feature	Hoos [14]	SMAC [16]	Matricon [21]	Our approach
Ranking/Scoring	✓	✗	(✓)	✓
Runtime Minimization	✗	✓	✓	✓
Incremental/Non-Random	✗	✗	✓	✓
Scalability	✓	✓	✗	✓

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 [26]. 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.

2.1 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]. Mısır presents a feature-based approach to reduce benchmarks by matrix factorization and clustering [23].

Hoos et al. [14] 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 [36, 37]. The approach does, however, not optimize for benchmark *runtime* and selects instances *randomly*, apart from constraints on the instance hardness and feature distance.

2.2 Algorithm Configuration

Further related work can be found within the field of algorithm configuration [15, 31], 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.

2.3 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 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.

2.4 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. [28] survey active-learning advances. While synthesis-based AL methods [5, 10, 33] generate instances for labeling, pool-based methods [12, 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, 10]. 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

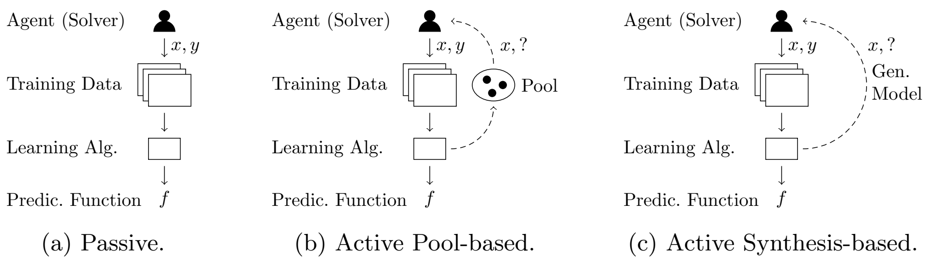


Fig. 1: Types of machine learning (depiction inspired by Rubens et.al. [28]).

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

pool-based AL to learn an instance-specific algorithm selector [34]. 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.

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 $f : \hat{\mathcal{A}} \times \mathcal{I} \to \mathbb{R}$ 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).

3.1.1 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 $\gamma_k : \mathcal{A} \times \mathcal{I} \to \{1, \dots, k\}$ 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, 25].

3.1.2 Ranking Solvers

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

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

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 [29]. These methods require the model's predictions to include probabilities for the k discrete runtime labels. Let $f' : \hat{\mathcal{A}} \times \mathcal{I} \to [0,1]^k$ denote this modified prediction function. In the following, the set $\tilde{\mathcal{I}} \subseteq \mathcal{I}$ denotes the instances that have already been sampled.

3.2.1 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:

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

3.2.2 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:

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

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

3.3.1 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 $\mathrm{w}(S, P)$ 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 $\hat{a}$:

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

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

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

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 $\big(s_k(a) - s_{\hat{a}}\big) \cdot \big(\text{par}_2(a) - \text{par}_2(\hat{a})\big) > 0$ 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     $O \leftarrow O \cup \big\{(O_{\text{acc}}, O_{\text{rt}})\big\}$
15 $\big(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}}\big) \leftarrow \text{average}(O)$
16 return $\big(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}}\big)$
```

3.3.2 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 $\hat{a}$.

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

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

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 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]. The dataset consists of 5355 instances with respective runtime data of 28 sequential SAT solvers. We also use a database of 56 instance features[3] from the Global Benchmark Database (GBD) by Iser et al. [17]. They comprise instance size features and node distribution statistics for several graph representations of SAT instances, among others, and are primarily inspired by the SATzilla 2012 features described in [37]. All features are numeric and free of missing values. We drop 10 out of 56 features because of zero variance. Overall, prediction models have access to 46 instance features and 27 runtime features, i.e., excluding the current new solver $\hat{a}$.

Additionally, we retrieve instance-family information[4] to evaluate the composition of our sampled benchmarks. Instance families comprise instances from the same application domain, e.g., planning, cryptography, etc., and are a valuable tool for analyzing solver performance.

For hyper-parameter tuning, we randomly sample 10 % of the complete set of 5355 instances with stratification regarding the instances' family. All instance families that

[3] https://benchmark-database.de/getdatabase/base_db
[4] https://benchmark-database.de/getdatabase/meta_db

are too *small*, i.e., 10 % of them corresponds to less than one instance, are put into one meta-family for stratification. This *tuning dataset* allows for a more extensive exploration of the hyper-parameter space.

4.3 Hyper-parameters

Given Algorithm 1, there are several possible instantiations for the three sub-routines, i.e., *ranking*, *selection*, and *stopping*. We describe these experimental configurations in the following.

4.3.1 Ranking

Regarding *ranking* (cf. Section 3.1), we experiment with the following approaches and hyper-parameter values:

- Observed PAR-2 ranking of already sampled instances
- Predicted runtime-label ranking
  - History size: Consider the latest 1, 10, 20, 30, or 40 predictions within a voting approach for stability. 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.

4.3.2 Selection

For *selection* (cf. Section 3.2), we experiment with the following methods and hyper-parameter values. Since the potential runtime of experiments is by magnitudes larger than the model's update time, we only consider incrementing our benchmark by one instance at a time rather than using batches, which is also proposed in current active-learning advances [30, 33]. A drawback of this 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 normalize uncertainty scores per instance by the average runtime of solvers on it or use the absolute values.
- 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 normalize information-gain scores per instance by the average runtime of solvers on it or use the absolute values.

Table 2: Regression performance of models trained on 46 instance and 27 runtime features. We report the RMSE and MAE metrics on the test sets.

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

4.3.3 Stopping

For *stopping* decisions (cf. Section 3.3), we experiment with the following criteria and hyper-parameter values:

- Subset-size stopping criterion, using 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 crucial parts of our framework is the choice of the surrogate model to predict runtime labels and inform active-learning selection decisions. In this section, we first detail why we do *not* directly predict absolute solver runtimes, which machine-learning models work best in predicting the time-outs of a solver, how clustering can be used to create more nuanced runtime labels, and how ML models perform on those cluster labels.

Table 2 shows the prediction performance of two regression models, a Random-Forest-Regressor and an MLP-Regressor, on the test set. Both models have been chosen because of their robustness and capability of capturing non-linear dependencies [3]. The instance feature vector consists of 46 instance features and 27 known solver runtimes. Both models perform poorly, which is evident from the magnitudes of the average regression errors. An RMSE error of about 2000 s is quite poor compared to the solver runtimes which range from 0 to 5000 s. Therefore, we decided to discretize runtimes.

Table 3 shows the performance of different models in predicting timeouts. While the second column shows the performance of models trained on the 46 instance features, the third column shows the performance of models trained on the 46 instance features and the 27 runtime features. While the Quadratic-DA model performs poorly on the 46

Table 3: MCC performance of models predicting timeouts. The models are trained 28 times with each solver as a target once.

Timeout Prediction Models	46 Instance Features Avg. MCC (± std)	46 + 27 Runtime Feat. Avg. MCC (± std)
Stacking (QDA + RF)	0.6513 (± 0.0458)	0.9527 (± 0.0292)
Quadratic-DA (QDA)	0.2593 (± 0.1310)	0.9290 (± 0.0339)
Random-Forest-Classifier (RF)	0.6607 (± 0.0547)	0.8530 (± 0.0479)
Ada-Boost	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-Classifier	0.5108 (± 0.0387)	0.7885 (± 0.1521)
MLP-Classifier	0.1293 (± 0.0718)	0.7760 (± 0.1408)
Support-Vector-Machines	0.0595 (± 0.0554)	0.7757 (± 0.2149)
Naive-Bayes	0.1173 (± 0.0872)	0.7306 (± 0.1394)

instance features, it performs well on the 46 instance features and 27 runtime features. The best prediction performance however is achieved by a stacking ensemble [35], which combines quadratic-discriminant analysis [32] and a random-forest classifier [3]. Stacking means that another prediction model, in our case a simple decision tree, decides which of the two ensemble members makes the prediction on which instance.

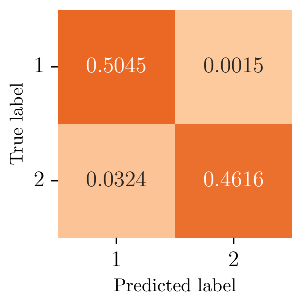
As already mentioned, runtime prediction is quite challenging. Instead, we discretize runtimes. To define prediction targets, we use hierarchical clustering with k clusters and a log-single-link criterion, which produced the most *useful* labels in preliminary experiments. We denote this adapted solver scoring function with s_k as defined in (1). In our chosen hierarchical procedure, each non-timeout runtime starts in a separate interval. We then gradually merge intervals whose single-link logarithmic distance is the smallest until the desired number of partitions is reached. The clustering is instance-specific, i.e., the boundaries are chosen such that it produces groups of solvers with similar performances for each instance. Other clustering approaches that we tried include hierarchical clustering with mean-, median-, and complete-link criterion, as well as k-means and spectral clustering.

We evaluated the performance of the best models from Table 3 in predicting these runtime clusters as mentioned above. Table 4 shows their performance in predicting the C_k clusterings for $k \in \{3, 4, 5\}$. While the second column shows the prediction performance of the models trained on the 46 instance features including the 27 known solver runtimes, the third column shows the prediction performance of the models trained on those features plus 27 features representing the known C_k clustering labels of the known solver runtimes. We observe that the models perform better when the clustering labels are included as features. Moreover, the prediction performance degrades rapidly with an increasing number of clusters. In all cases, the stacking ensemble performs best.

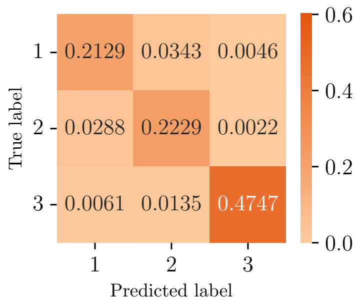
Our final choice for the prediction model is the stacking ensemble of a quadratic-discriminant analysis and a random forest. Both these models can learn non-linear relationships between the instance features and the runtime labels. Figure 2 shows the

Table 4: MCC performance of the best models predicting the C_k clusterings.

C_3 Cluster Prediction Models	$46+27$ Runtime Feat. Avg. MCC ($\pm$ std)	$46+27+27$ Clus. Feat. Avg. MCC ($\pm$ std)
Stacking (QDA + RF)	0.7464 ($\pm$ 0.0497)	0.8380 ($\pm$ 0.0634)
Quadratic-DA (QDA)	0.6903 ($\pm$ 0.0607)	0.7738 ($\pm$ 0.0459)
Random-Forest-Classifier (RF)	0.7116 ($\pm$ 0.0385)	0.7841 ($\pm$ 0.0469)
C_4 Cluster Prediction Models	**$46+27$ Runtime Feat. Avg. MCC ($\pm$ std)**	**$46+27+27$ Clus. Feat. Avg. MCC ($\pm$ std)**
Stacking (QDA + RF)	0.6562 ($\pm$ 0.0538)	0.7366 ($\pm$ 0.0377)
Quadratic-DA (QDA)	0.6007 ($\pm$ 0.0765)	0.6872 ($\pm$ 0.0469)
Random-Forest-Classifier (RF)	0.6033 ($\pm$ 0.0455)	0.6828 ($\pm$ 0.0527)
C_5 Cluster Prediction Models	**$46+27$ Runtime Feat. Avg. MCC ($\pm$ std)**	**$46+27+27$ Clus. Feat. Avg. MCC ($\pm$ std)**
Stacking (QDA + RF)	0.5835 ($\pm$ 0.0615)	0.6396 ($\pm$ 0.0553)
Quadratic-DA (QDA)	0.5508 ($\pm$ 0.0669)	0.5881 ($\pm$ 0.0912)
Random-Forest-Classifier (RF)	0.5420 ($\pm$ 0.0459)	0.5895 ($\pm$ 0.0475)



(a) Timeout prediction.



(b) C_3 cluster prediction.

Fig. 2: Confusion matrices for predicting timeouts and C_3 clusters.

confusion matrices of the stacking ensemble for predicting timeouts and C_3 clusters. Both perform reasonably well, with the C_3 cluster prediction model performing slightly worse than the timeout prediction model. Notably, for the C_3 prediction model, 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 indicates that deciding timeouts is easier than distinguishing different groups of solvers.

To obtain *useful* labels, we also need to ensure that the discretized labels still discriminate solvers and align 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 runtime score and according to our discretized s_3 score. We observe that our ranking approach correctly decides almost all (about 97.45 %; σ = 3.68 %) solver pairs, i.e., which solver is faster among two given solvers. In particular,

Table 5: Ranking of Anniversary-Track solvers with standard PAR-2 runtime and our discretized s_3 ranking.

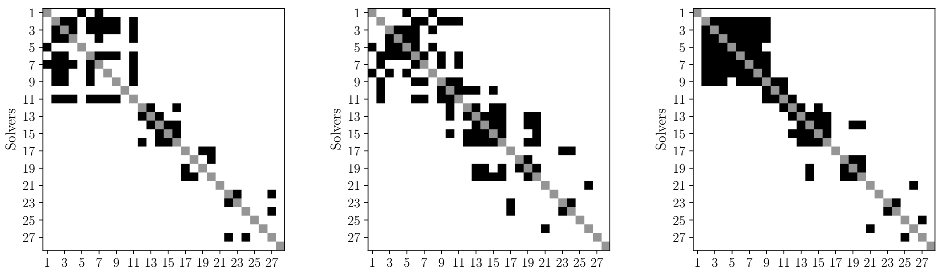
PAR-2		s_3		Solver
Rank	Score	Rank	Score	
1	2808.13	1	1.1717	kissat-mab-esa
2	2812.93	2	1.1832	kissat-sc2022-bulky
3	2835.25	3	1.1862	ekissat-mab-gb-db
4	2835.59	4	1.1868	kissat-mab-ucb
5	2836.92	5	1.1868	kissat-inc
6	2845.19	6	1.1926	ekissat-mab-db-v1
7	2846.73	7	1.1930	kissat-mab-moss
8	2857.67	8	1.1947	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

the Spearman correlation of s_3 and PAR-2 ranking is about 0.988, which is very close to the optimal value of 1 [6].

We further analyzed how runtime discretization affects the distinguishability of solver pairs. Figure 3 shows the confusion matrices of solver pairs with significant differences in their PAR-2 scores, timeouts, and s_3 scores. According to a Wilcoxon signed-rank test with level $\alpha = 0.05$, 87.83 % of solver pairs have significantly different scores after discretization, only a slight drop compared to 89.95 % before discretization. These results underpin the usefulness of discretized runtimes for our framework.

4.5 Implementation Details

For reproducibility, our source code and data are available on GitHub (cf. footnotes in Section 1). Our code is implemented in Python using *scikit-learn* [27] for making predictions and *gbd-tools* [17] for SAT-instance retrieval.



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

Fig. 3: Solver pairs with significant differences regarding the respective scorings are colored white. Non-significant differences are colored black. Solvers are ordered indentically to Tab. 5.

5 Evaluation

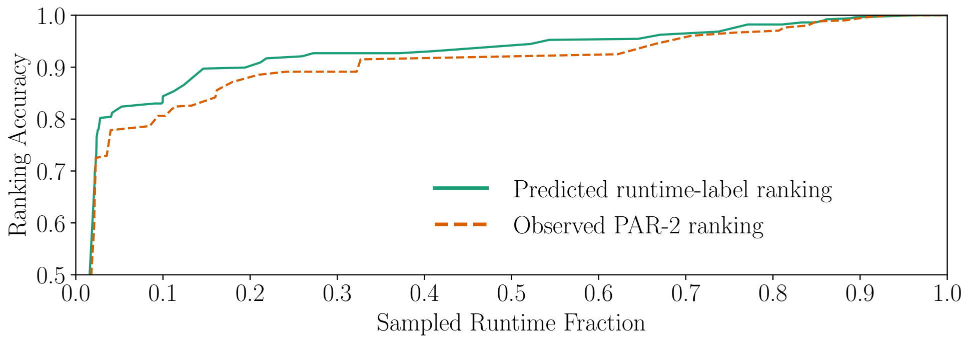
In this section, we evaluate our active-learning framework. First, we analyze and tune the different sub-routines of our framework on the tuning dataset. Next, we evaluate the best configurations with the full dataset. Finally, we analyze the importance of different instance families to our framework.

5.1 Hyper-Parameter Analysis

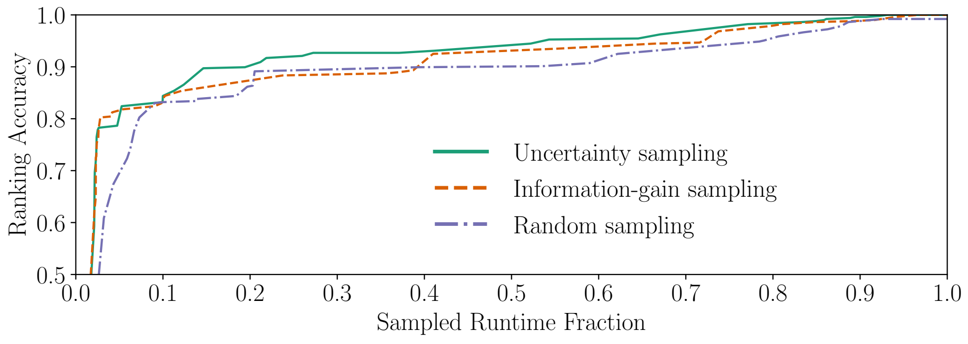
Our experiments follow the evaluation framework introduced in Section 4.1. Fig. 4 shows the performance of the approaches from Section 4.3 on O_{rt}-O_{acc}-diagrams for the hyper-parameter-tuning dataset. Evaluating a particular configuration with Algorithm 2 returns a point (O_{rt}, O_{acc}). We do 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., of all configurations that share a particular value of the plotted hyper-parameter, we take the maximum ranking accuracy over all remaining hyper-parameters *not* displayed in the corresponding plot.

Regarding ranking approaches (Fig. 4a), using the predicted s_3-induced runtime-label ranking consistently outperforms the partially observed PAR-2 ranking for each possible value of the trade-off parameter δ. This outcome is expected since selection decisions are not random. For example, we might sample more instances of one family if it benefits discrimination of solvers. While the partially observed PAR-2 score is skewed, the prediction model can account for this.

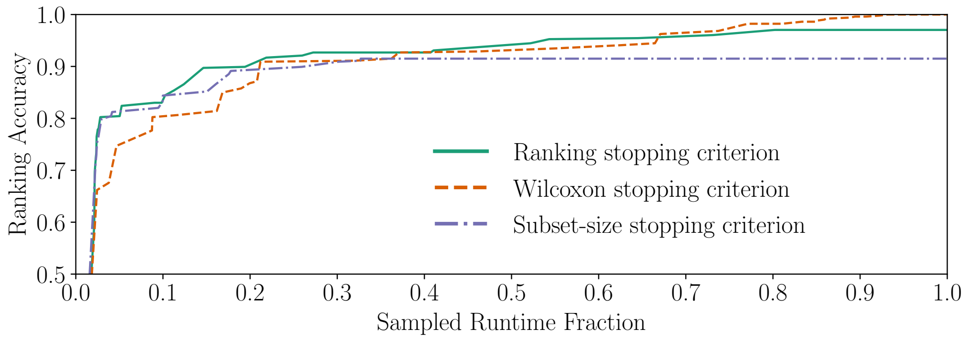
Regarding the selection approaches (Fig. 4b), uncertainty sampling performs best in most cases. However, information-gain sampling is beneficial if runtime is strongly favored (small δ; runtime fraction less than 5 %). This result aligns with our expectations: Information-gain sampling selects instances that maximize the expected reduction in entropy. This means we sample instances revealing similarities between



(a) Ranking approaches

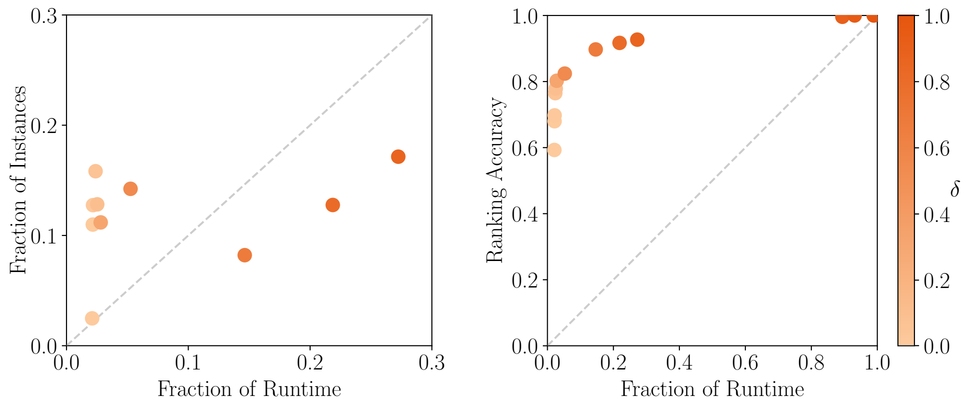


(b) Selection approaches



(c) Stopping criteria

Fig. 4: O_{rt}-O_{acc}-diagrams comparing different hyper-parameter instantiations of our active-learning framework on the hyper-parameter-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 (cf. Section 4.1). Each line entails the front of Pareto-optimal configurations for the respective hyper-parameter instantiation.



(a) Runtime vs. Instances (b) Runtime vs. Accuracy

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

solvers rather than differences, which helps to build a confident model quickly. However, the method cannot select helpful instances for distinguishing solvers later. Random sampling performs reasonably well but is outperformed by uncertainty sampling in all cases, showing the benefit of actively selecting instances based on a prediction model.

Regarding the stopping criteria (Fig. 4c), the ranking stopping criterion performs most consistently well. If 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 of sampling a fixed subset of instances.

Fig. 5a shows an interesting consequence of weighting our optimization goals: If we, on the one hand, desire to get a *rough* estimate of a solver's performance fast (low δ), approaches favor 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. If we, on the other hand, desire to get a *good* estimate of a solver's performance in a moderate amount of time (high δ), approaches favor selecting few, *difficult* instances. In particular, the fraction of instances is smaller than the fraction of runtime.

Furthermore, Fig. 5b reveals which values make the most sense for δ. The range $\delta \in [0.2, 0.8]$, thereby, corresponds to the points with a runtime fraction between 0.03 and 0.22 We consider this region to be most promising, analogous to the *elbow* method in cluster analysis [18].

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 statically selecting 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

5.2 Full-Dataset Evaluation

Having selected the most promising hyper-parameters, we run our active-learning experiments on the complete Anniversary Track dataset (5355 instances). The aforementioned range $\delta \in [0.2, 0.8]$ only results in two distinct 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 a new solver's PAR-2 ranking with 90.48 % accuracy (O_{acc}) in only 5.41 % of the full 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 can predict a new solver's PAR-2 ranking with 92.33 % accuracy (O_{acc}) in only 10.35 % of the full evaluation time (O_{rt}).

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

Both our AL approaches perform better than random sampling. However, the performance differences are not significant regarding a Wilcoxon signed-rank test with $\alpha = 0.05$ and also depend on the fraction of sampled runtime (cf. Fig. 4b). A clear advantage of our approach is, though, that it indicates when to stop adding further instances, depending on the trade-off parameter δ. While the active-learning results are less strong on the full dataset than on the smaller tuning dataset, they still show the benefit of making benchmark selection dependent on the solvers to distinguish.

A static benchmark using the most frequently AL-sampled instances performs poorly, though, compared to active learning and random sampling. This outcome is somewhat expected since the static benchmark does not reflect the right balance of instance families: Families whose instances are uniform-randomly selected by AL, e.g., for different solvers, appear less often in this benchmark than families where some instances are sampled more often than others.

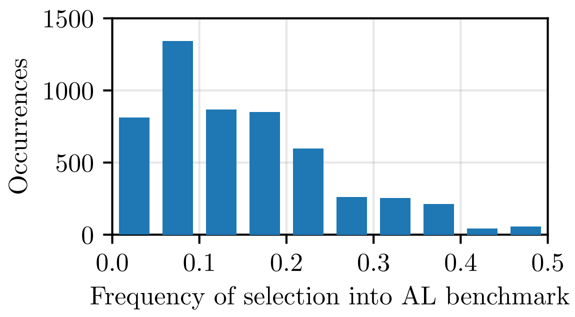


Fig. 6: Frequency of how often individual problem instances are selected within all active-learning benchmarks. Instances that are chosen more frequently are intuitively more important to distinguish solver runtimes.

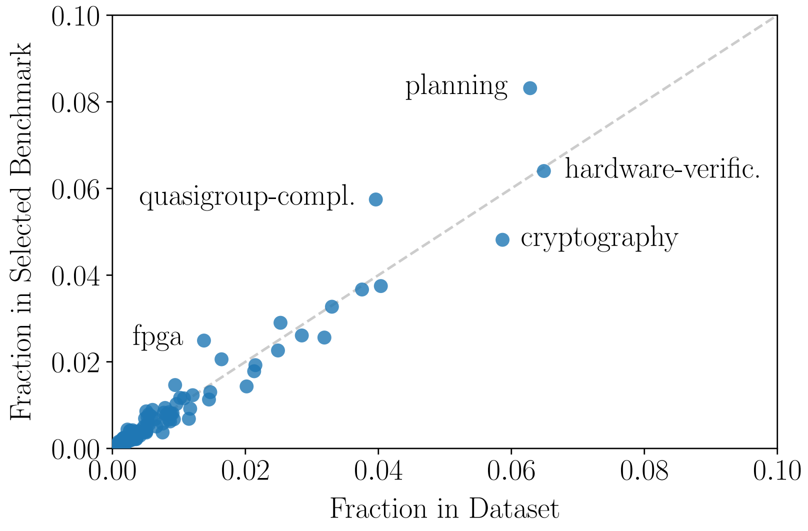


Fig. 7: Scatter plot showing the *importance* of different instance families to our framework on the full dataset. The x-axis shows the frequency of instance families in the dataset. The y-axis shows the average frequency of instance families in the samples selected by active learning. The dashed line represents families that occur with the same frequency in the dataset and samples.

5.3 Instance Frequency and Instance-Family Importance

We analyze the frequency of individual instances being selected by our active-learning approach. Fig. 6 shows the distribution of how often individual instances are selected within all active-learning benchmarks. We observe that a small fraction of instances is selected very often, while most instances are selected only a few times. There is no fixed subset of instances that can perfectly distinguish all solvers, which shows the usefulness of employing an active-learning-based strategy.

Selection decisions of our approach also reveal the importance of different instance families to our framework. Fig. 7 shows the occurrence of instance families within the dataset and the benchmarks created by active learning. We use the best-performing configurations for all $\delta \in [0, 1]$ and examine the selection decisions by the active-learning approach on the SAT Competition 2022 Anniversary Track dataset [2]. While most families appear with the same fraction in the dataset and the sampled benchmarks, a few outliers need further discussion. Problem instances of the families *fpga*, *quasigroup-completion*, and *planning* are especially helpful to our framework in distinguishing solvers. Instances of these families are selected over-proportionally in comparison to the full dataset. In contrast, instances of the largest family, i.e., *hardware-verification*, roughly appear with the same fraction in the dataset and the sampled benchmarks. Finally, instances of the family *cryptography* are less important in distinguishing solvers than their vast weight in the dataset suggests. A possible explanation is that these instances are very 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 want to find its ranking amidst competitors. Our approach provides accurate ranking predictions while needing 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 needing 10 % of the full-evaluation time. We have evaluated several ranking algorithms, instance-selection approaches, and stopping criteria within our sequential active-learning framework. We also took a brief look at which instance families are the most prevalent in selection decisions.

Future work may compare further sub-routines for ranking, instance selection, and stopping. Additionally, one can apply our evaluation framework to arbitrary computation-intensive problems, e.g., other $\mathcal{NP}$-complete problems than SAT, as all discussed active-learning methods are problem-agnostic. Such problems share most of the relevant properties of SAT solving, i.e., there are established instance features, a complete benchmark is expensive, and traditional benchmark selection requires expert knowledge.

From the technical perspective, one could formulate 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 is unclear 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. Artif Intell 223:120–155. https://doi.org/10.1016/j.artint.2015.01.002

[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. Mach Learn 45(1):5–32. https://doi.org/10.1023/A:1010933404324

[4] Collautti M, Malitsky Y, Mehta D, et al (2013) SNNAP: solver-based nearest neighbor for algorithm portfolios. In: Proc. ECML PKDD, pp 435–450, https://doi.org/10.1007/978-3-642-40994-3_28

[5] Dang N, Akgün Ö, Espasa J, et al (2022) A framework for generating informative benchmark instances. In: Proc. CP, pp 18:1–18:18, https://doi.org/10.4230/LIPIcs.CP.2022.18

[6] 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. Psychol Methods 21(3):273–290. https://doi.org/10.1037/met0000079

[7] Dehghani M, Tay Y, Gritsenko AA, et al (2021) The benchmark lottery. arXiv:210707002 [csLG] URL https://arxiv.org/abs/2107.07002

[8] Froleyks N, Heule M, Iser M, et al (2021) SAT Competition 2020. Artif Intell 301. https://doi.org/10.1016/j.artint.2021.103572

[9] Fuchs T, Bach J, Iser M (2023) Active learning for SAT solver benchmarking. In: Proc. TACAS, pp 407–425, https://doi.org/10.1007/978-3-031-30823-9_21

[10] Garzón I, Mesejo P, Giráldez-Cru J (2022) On the performance of deep generative models of realistic SAT instances. In: Proc. SAT, pp 3:1–3:19, https://doi.org/10.4230/LIPIcs.SAT.2022.3

[11] Gelder AV (2011) Careful ranking of multiple solvers with timeouts and ties. In: Proc. SAT, pp 317–328, https://doi.org/10.1007/978-3-642-21581-0_25

[12] Golbandi N, Koren Y, Lempel R (2011) Adaptive bootstrapping of recommender systems using decision trees. In: Proc. WSDM, pp 595–604, https://doi.org/10.1145/1935826.1935910

[13] Harpale A, Yang Y (2008) Personalized active learning for collaborative filtering. In: Proc. SIGIR, pp 91–98, https://doi.org/10.1145/1390334.1390352

[14] Hoos HH, Kaufmann B, Schaub T, et al (2013) Robust benchmark set selection for boolean constraint solvers. In: Proc. LION, pp 138–152, https://doi.org/10.1007/978-3-642-44973-4_16

[15] 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, https://doi.org/10.3233/FAIA200995

[16] Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. In: Proc. LION, pp 507–523, https://doi.org/10.1007/978-3-642-25566-3_40

[17] Iser M, Sinz C (2018) A problem meta-data library for research in SAT. In: Proc. PoS, pp 144–152, https://doi.org/10.29007/gdbb

[18] Kodinariya TM, Makwana PR (2013) Review on determining number of cluster in k-means clustering. Int J Adv Res Comput Sci Manage Stud 1(6):90–95. URL http://www.ijarcsms.com/docs/paper/volume1/issue6/V1I6-0015.pdf

[19] Koren Y, Bell RM, Volinsky C (2009) Matrix factorization techniques for recommender systems. Computer 42(8):30–37. https://doi.org/10.1109/MC.2009.263

[20] Manthey N, Möhle S (2016) Better evaluations by analyzing benchmark structure. In: Proc. PoS, URL http://www.pragmaticsofsat.org/2016/reg/POS-16_paper_4.pdf

[21] Matricon T, Anastacio M, Fijalkow N, et al (2021) Statistical comparison of algorithm performance through instance selection. In: Proc. CP, pp 43:1–43:21, https://doi.org/10.4230/LIPIcs.CP.2021.43

[22] Mısır M (2017) Data sampling through collaborative filtering for algorithm selection. In: Proc. IEEE CEC, pp 2494–2501, https://doi.org/10.1109/CEC.2017.7969608

[23] Mısır M (2021) Benchmark set reduction for cheap empirical algorithmic studies. In: Proc. IEEE CEC, pp 871–877, https://doi.org/10.1109/CEC45853.2021.9505012

[24] Mısır M, Sebag M (2017) ALORS: An algorithm recommender system. Artif Intell 244:291–314. https://doi.org/10.1016/j.artint.2016.12.001

[25] Ngoko Y, Cérin C, Trystram D (2019) Solving SAT in a distributed cloud: A portfolio approach. Int J Appl Math Comput Sci 29(2):261–274. https://doi.org/10.2478/amcs-2019-0019

[26] 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 Min Knowl Discov 12(2). https://doi.org/10.1002/widm.1441

[27] Pedregosa F, Varoquaux G, Gramfort A, et al (2011) Scikit-learn: Machine learning in Python. J Mach Learn Res 12(85):2825–2830. URL http://jmlr.org/papers/v12/pedregosa11a.html

[28] 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, https://doi.org/10.1007/978-1-4899-7637-6_24

[29] 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

[30] Sinha S, Ebrahimi S, Darrell T (2019) Variational adversarial active learning. In: Proc. ICCV, pp 5971–5980, https://doi.org/10.1109/ICCV.2019.00607

[31] Stützle T, López-Ibáñez M, Pérez-Cáceres L (2022) Automated algorithm configuration and design. In: Proc. GECCO, pp 997–1019, https://doi.org/10.1145/3520304.3533663

[32] Tharwat A (2016) Linear vs. quadratic discriminant analysis classifier: a tutorial. Int J Appl Pattern Recognit 3(2):145–180. https://doi.org/10.1504/IJAPR.2016.079050

[33] Tran T, Do T, Reid ID, et al (2019) Bayesian generative active deep learning. In: Proc. ICML, pp 6295–6304, URL http://proceedings.mlr.press/v97/tran19a.html

[34] Volpato R, Song G (2019) Active learning to optimise time-expensive algorithm selection. arXiv:190903261 [csLG] URL https://arxiv.org/abs/1909.03261

[35] Wolpert DH (1992) Stacked generalization. Neural Networks 5(2):241–259. https://doi.org/10.1016/S0893-6080(05)80023-1

[36] Xu L, Hutter F, Hoos HH, et al (2008) SATzilla: Portfolio-based algorithm selection for SAT. J Artif Intell Res 32:565–606. https://doi.org/10.1613/jair.2490

[37] 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