Active Learning for SAT-Solver Benchmarking

Tobias Fuchs[0000−0001−9727−2878] (✉),
Jakob Bach[0000−0003−0301−2798], and
Markus Iser[0000−0003−2904−232X]

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
info@tobiasfuchs.de, jakob.bach@kit.edu, markus.iser@kit.edu

Abstract. Experimentation is one of the main phases in algorithm engineering, including the propositional-satisfiability problem (SAT). While the field of SAT-solver benchmarking is well established, traditional benchmark-set-selection approaches do not optimize for benchmark runtime. Benchmark-set selection is the process of choosing a subset of instances out of a pool to be representative and reliable in discriminating SAT solvers. This paper presents an active-learning-based benchmark-set-selection approach that solves the *New-Solver Problem*: Given a new solver, we want to find its ranking amidst its competitors with as little runtime as possible while still giving accurate ranking predictions. We can determine a new solver's rank regarding the SAT Competition 2022 Anniversary Track instances in about 10 % of the time it would take to run the solver on all instances of this dataset. We achieve this while still maintaining a prediction accuracy of about 92 % regarding the PAR-2 ranking of this solver if it were evaluated on the complete dataset. Additionally, we discuss the importance of instance families in the selection process. Our tool provides the engineer of propositional-satisfiability solvers with a faster way of determining a new SAT-solver's performance in comparison to sampling the benchmark set by hand.

Keywords: Propositional satisfiability · Benchmarking · Active learning

1 Introduction

One of the main phases of algorithm engineering is experimentation. This is also true for the propositional-satisfiability problem (SAT), the archetypal $\mathcal{NP}$-complete problem. While state-of-the-art solvers embark upon more and more problem instances, SAT-solver evaluation is steadily getting more costly. Usually, solvers submitted to competitions like the annual SAT Competition [9,2] themselves precede heavy experimentation and are the remnant of a multitude of ideas and considerations. While the field of SAT-solver benchmarking is well established, traditional benchmark-set-selection approaches [11,18] do not optimize for benchmark runtime. Their goal is to select a representative and robust benchmark set for discriminating solvers reliably. Within our work, we show that we can arrive at the same conclusions with high accuracy in only a fraction of the time that would be needed to evaluate the full benchmark set. To be more specific, we address the following problem:

Definition 1 (New-Solver Problem). *Given a pool of instances $\mathcal{I}$ and solvers $\mathcal{A}$ with already known runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau] \cup \{\perp\}$ (timeout τ usually 5000 s [9]), the problem is to decide with as little resources (CPU time) as possible how a new solver $\hat{a} \notin \mathcal{A}$ performs in comparison to the existing ones by incrementally selecting a benchmark set $\tilde{\mathcal{I}}$ on which to evaluate $\hat{a}$.*

Note that we know the runtimes of all solvers, except the new one, on all instances. There also is work on collaborative filtering, where runtimes are only partially known, for algorithm selection [30,28]. We want to find a scoring function $\text{score} : \mathcal{A} \cup \{\hat{a}\} \to \mathbb{R}$ assigning each solver a score. SAT Competitions [9] typically rely on the PAR-2 score, i.e, the average runtime with a penalty term of 2τ for timeouts. We predict the solver ranking induced by this scoring, without predicting exact solver runtimes. This allows for both comparing the solvers as well as identifying solvers with similar behavior if looking at individual predictions per instance.

This current article is proposing a novel active-learning-based benchmark-set-selection approach. Our algorithm is unique in that it is the only one, to the best of our knowledge, that has all of the following features (also refer to Tab. 1). Our benchmarking process outputs a *ranking* to compare the new solver against the existing set of solvers as described above. We optimize the *runtime* that our strategy needs to arrive at its conclusion. We use problem-instance and runtime *features* $\varphi : \mathcal{A} \times \mathcal{I} \to \mathbb{R}^n$ since such features already have proven to be helpful in classic benchmark-set-selection approaches [18] as well as in portfolio selection [4,31]. Moreover, we select instances *non-randomly* and *incrementally*. By doing so, we have a greater impact on the properties of the benchmarking approach such as its required runtime. Sampling instances incrementally means taking runtime information from already done experiments into account when choosing the next. Rather than solely outputting a binary classification, i.e, the new solver is worse than solver A, we provide a *scoring function* that shows by which margin a solver is worse and how similar it is to existing approaches (*comparability*). Finally, our strategy incrementally selects only a single benchmark set to compare one new solver $\hat{a}$ with the set of existing solvers $\mathcal{A}$ whose cardinality can be any number (*scalability*). We avoid comparing pairs of solvers, which requires a non-constant number of benchmark sets. Since a new solver idea is rarely best on the first try but requires more or less tweaking, it is desired to obtain a solver ranking fast. In this way, a new solver idea can be discarded if it performs poorly across the board or may be further tweaked if it shows promising results at least in some cases.

To sum up our contribution, we take the SAT Competition 2022 Anniversary Track benchmark [2] (5355 instances, full runtimes of 28 solvers), perform cross-validation by treating each solver as new once, and predict these solvers' PAR-2 rank with about 92 % accuracy with only about 10 % of the runtime that would be needed to evaluate these solvers on the complete set of instances. Both results are averages across all cross-validation iterations.

For reproducibility, all our source code[1] and and data[2] are available on GitHub.

2 Related Work

This section discusses work in several related fields. First, we take a look at traditional SAT-solver benchmarking. Second, we explore how SAT-solver configuration systems deal with navigating possible configurations. Third, we discuss an existing benchmark-set-selection approach that optimizes for benchmark runtime. Finally, we show how active-learning techniques can help to tackle the aforementioned *New-Solver Problem*.

Tab. 1 shows a concise overview of existing approaches with a related goal. While traditional benchmark-set-selection [18] lacks the aspect of runtime optimization, related work within algorithm configuration [19] is concerned with finding better configurations fast. However, algorithm configuration is not suited for ranking. There is already existing work employing active learning for SAT-solver discrimination [25]. This approach has some major shortcomings though. Finally, our approach seeks to provide all stated features. Note that all approaches already make use of problem-instance features as it showed to be beneficial in past work [18,4,31].

Traditional benchmark-set-selection. Recent studies [7,32] show that selecting a benchmark set in general is a difficult endeavor for multiple reasons. Dehghani et. al. [7] describe how a biased benchmarking set can easily lead to fallacious interpretations. Moreover, Nießl et. al. [32] state that benchmark-set design has many movable parts, e.g., performance measures, aggregation, and handling of missing values. Questionable research practices might modify those design elements a-posteriori to fit expectations, leading to biased results.

Benchmark-set selection is an important issue for solver competitions, e.g., the SAT Competition. In such competitions, the organizers define the rules for composing the corresponding benchmarks. Balint et al. [1] provide an overview of benchmark-selection criteria in different solver competitions. Manthey and Möhle [24] find that competition benchmarks might contain redundant instances, and propose a feature-based approach to remove redundancy. Mısır [29] presents a feature-based approach to reduce benchmark sets by matrix factorization and clustering.

Hoos et. al. [18] discuss which properties are most desirable when selecting a set of SAT benchmark instances. The selection strategy presented is static, i.e., it does not depend on particular solvers to distinguish. Desirable properties are a variety of instances (avoids over-fitting of solver approaches), adapted instance hardness (not too easy but also not too hard; solvers cannot be distinguished if too many time-out or finish immediately), and no duplicates (having duplicates

[1] temporary, anonymized version for review: xxx
[2] temporary, anonymized version for review: xxx

Table 1: Support for desired features that are introduced in Sec. 1. The shown approaches include traditional benchmark-set-selection techniques [18], an algorithm configuration system [19], an existing active-learning approach [25], as well as our approach.

Feature	Hoos et. al. [18]	SMAC [19]	Matricon et. al. [25]	Our approach
Ranking	✓	✗	✓	✓
Runtime optimization	✗	✓	✓	✓
Feature-based	✓	✓	✓	✓
Incremental non-random sampling	✗	✗	✓	✓
Comparability	✓	✗	✗	✓
Scalability	✓	✓	✗	✓

gives those instances more weight). To avoid having too similar instances, they make use of a distance-based approach using the SATzilla features [41,42]. Those features comprise general instance-size features and graph-representation features among others. Their approach *ranks* solvers by providing its PAR-2 score, is *feature-based*, and is *scalable* in the sense that the evaluation time scales linearly with more instances and solvers. Also, solvers are *comparable* within their approach in the sense that the obtained PAR-2 score induces a distance between solvers. However, they do not optimize for benchmark *runtime* and select instances *randomly*, apart from constraints on the instance hardness and feature distance.

SAT-solver configuration. Another interesting approach can be found within algorithm-configuration systems [19,17,37]. The goal is to tune SAT solvers for a given sub-domain of problem instances.

SMAC [19] tries to find a close-to-optimal configuration for a given SAT-solver on a given set of instances. Although this task is different from our goal, e.g., we do not need to navigate configuration space, they use a similar framework. There are mainly two similarities. First, both approaches are model-based. While SMAC tries to gradually build a model of the solver's performance as a function of the configuration parameters, we want to gradually build a model of a new solver's performance based on already-known runtimes of instances and runtime information on already-known solvers. Second, both approaches employ some kind of selection strategy that is based on the previously mentioned model. Theirs selects configurations rather than instances though. Furthermore, Hutter et. al. [19] use an approach that makes use of exploitation (what configuration parameters are already good) and exploration (parameter space with few data points) to select the next configuration for which to acquire runtime information. SMAC's configuration sampling has also proven to be better than random sampling in all tested scenarios and has even proven to be significantly better in most cases [19].

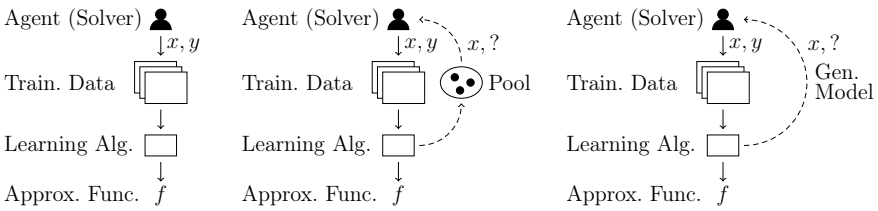
While this approach considers the configuration's *runtime*, is *feature-based*, and *scales linearly* by having a fixed time budget, it is not capable of the other three features listed in Tab. 1. First, an algorithm configurator cannot be used to *rank* a new solver since it only looks at promising solvers/configurations rather than the overall average performance. If the new solver does not belong to the best-performing solvers (new ideas are rarely the best approach right from the start, i.e., they need some tweaking), we do not have any information on how this solver performed. Second, while using a model-based selection strategy to sample configurations, instance sampling is done *randomly*, i.e., without building a model over instances. In fact, tuning algorithms to a small or non-homogeneous set of instances might lead to over-tuning, i.e., lack of generalization [8]. And third, configurations within their approach are not *comparable* since algorithm configuration seeks to find solemnly the best performing configuration.

Incremental benchmark-set-selection. Matricon et. al. [25] present an incremental benchmark-set-selection approach. Their *per-set efficient algorithm selection problem* (PSEAS) is similar to our *New-Solver Problem* (cf. Definition 1). The main difference is that Matricon et. al. compare a solver to another one, while we compare it to a set of existing solvers. To the best of our knowledge, we are the first to tackle the latter problem.

Given a pair of SAT solvers, Matricon et. al. want to iteratively select a subset of instances to decide with high confidence which of the two solvers is better. The selection of instances is dependent on the choice of the solvers to distinguish. They calculate a scoring metric for all instances (that have not been selected yet), run the experiment with the highest score, and update the confidence. If the confidence exceeds a threshold (e.g., 95%), they stop iterating. Most of the proposed strategies in [25] are instance-based methods, i.e., no model of the solver runtimes is built.

For the scoring metric, they discuss (1) a random strategy as a baseline, (2) a discrimination strategy based on the work of Gent et. al. [12], (3) a variance-based strategy (select instance with the highest variance in its distribution of running times), (4) an information-gain-based method, and (5) a feature-distance-based strategy. To update the confidence level after each experiment, they discuss using (1) a fixed subset size, (2) a Wilcoxon test, and (3) a distribution-based method. Their experiments show that random instance selection with a Wilcoxon test as the stopping criterion performs consistently well across different data sets in the sense that it belongs to the Pareto front or is close to it in all cases.

Regarding our desired features of adaptive benchmark-set-selection (Tab. 1), the aforementioned approach already ticks off the first four features, i.e., it *ranks* solvers, optimizes for *runtime*, is *feature-based*, and uses *incremental non-random* sampling. Although they provide a confidence metric for their prediction, which makes solvers partially *comparable*, it is not clear how similar two given solvers are or on which instances they behave similarly, as the prediction is solely binary. Moreover, the major shortcoming lies within it being not *scalable* with the number solvers. As they compare only pairs of solvers, evaluating a new solver



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

Fig. 1: Types of Learning (depiction inspired by Rubens et.al. [34])

requires a non-constant number of benchmark sets, which in turn requires more runtime.

Active learning (AL). The posed *New-Solver Problem* has stark similarities to the well-studied field of active learning within recommender systems, especially the *new-user problem* [34]. On the one hand, we want to maximize the utility an instance provides to our model and, on the other hand, minimize the cost (CPU time) that is associated with its acquisition.

AL algorithms can be categorized into *synthesis-based* [5,10,39] and *pool-based* approaches [13,16,22].While synthesis-based methods (Fig. 1c) generate instances that might be interesting to label, pool-based methods (Fig. 1b) rely on a pool of unlabeled instances from which to draw items.

Recent synthesis-based methods within the field of satisfiability solving [5,10] show how to generate problem instances with desired properties. This goal is, however, orthogonal to ours. While those approaches want to generate problem instances on which a solver is good or bad, we want to predict whether a solver is good or bad on an existing benchmark set.

Pool-based methods can be further refined into *Error-based* [13,22], *Uncertainty-based* [16,21,38], *Distribution-based* methods [15,36] and *Ensembles* [23,27]. Error-based strategies pick instances that have the most influence on the model parameters. Uncertainty-based methods sample instances close to their perceived decision boundary. Distribution-based approaches sample instances whose features are unaccounted for amongst the already sampled instances. Finally, ensembles are a meta-strategy that combines the predictions of several different active-learning models into a selection choice.

In the field of SAT-solving, Volpato and Guangyan [40] use uncertainty-sampling-based AL to learn an instance-specific algorithm selector. Rather than benchmarking a solver's overall performance, the goal is to recommend the best solver out of a set of solvers for each SAT instance.

Summary. To sum up this section, we have shown both traditional and recent benchmark-set-selection approaches. Moreover, we have briefly outlined how active-learning techniques can help to incrementally select benchmark sets. In the

following sections, we present an active-learning-based benchmark-set-selection approach that solves the *New-Solver Problem* while optimizing for runtime and fulfilling both comparability and scalability.

3 Active Learning for SAT-Solver Benchmarking

Our framework follows a simple three-step schema (Alg. 1). After initializing the model $\mathcal{M}$, we incrementally *select* a problem instance from the pool of instances, one at a time, evaluate the new solver $\hat{a}$ on it, and *update* the model with the acquired result. We *stop* when meeting some stopping condition. Since the potential runtime of experiments is by magnitudes larger than the model's update time, we only consider incrementing our benchmark set by one instance at a time rather than using batches. At the core of the model $\mathcal{M}$ is a prediction function $f\colon (\mathcal{A} \cup \{\hat{a}\}) \times \mathcal{I} \to \mathbb{R}$ that powers the decisions within the previously mentioned three steps.

Algorithm 1: Incremental benchmarking framework

`Input:` A pool of instances $\mathcal{I}$, a set of known solvers $\mathcal{A}$ with known runtimes $r\colon \mathcal{A} \times \mathcal{I} \to [0, \tau] \cup \{\perp\}$, and a new solver $\hat{a} \notin \mathcal{A}$.

1 $\mathcal{M} \leftarrow$ init model with $\mathcal{A}$, $\mathcal{I}$, r, and $\hat{a}$
2 $\tilde{\mathcal{I}} \leftarrow \emptyset$
3 `while` not stop($\mathcal{M}$) `do`
4 $\quad s \leftarrow$ selectNextInstance($\mathcal{M}$)
5 $\quad t \leftarrow$ runExperiment($\hat{a}$, s) // *Runs $\hat{a}$ on s with time-out τ*
6 $\quad \tilde{\mathcal{I}} \leftarrow \tilde{\mathcal{I}} \cup \{s\}$
7 $\quad$ updateModel($\mathcal{M}$, s, t)
8 score() $\leftarrow$ predictScores($\mathcal{M}$)

`Output:` Return tuple of scoring function score$\colon \mathcal{A} \cup \{\hat{a}\} \to \mathbb{R}$ and the selected instances $\tilde{\mathcal{I}}$.

The following section (Sec. 3.1) discusses how we may derive a solver ranking from our model (line 8). Thereafter, we look at possible sampling criteria (line 4) in Sec. 3.2 and, finally, possible stopping conditions (line 3) in Sec. 3.3.

3.1 Predicting Rankings

In order to predict a ranking of solver $\hat{a}$ amidst its competitors, we need to infer its performance on all instances, even the ones on which we have not evaluated it. To do so, there are generally two notions of this problem, i.e., regression and classification. A regression model tries to predict the PAR-2 runtime of $\hat{a}$ on all instances based on the samples that have been selected. Then, the ranking is induced by the average predictions. Alternatively, a classification model can be

used to predict some discrete runtime labels of $\hat{a}$ on all instances. Similarly, the average runtime label determines the ranking.

Preliminary experiments, as well as research in portfolio solvers [31,4], have shown however that directly predicting runtimes without any transformation does not work well in practice. In contrast to that, discretizing runtimes and using those labels in the model's training and prediction yields good results (cf. Sec. 5). The following paragraphs motivate why it makes sense to discretize runtimes.

We use a function $\gamma\colon \mathcal{A} \times \mathcal{I} \to \{1, \ldots, k\}$ to transform runtime observations. In our experiments, this function is a clustering algorithm. Note that this transformation is done per instance to adapt to different instance difficulties. Also, we define $\gamma(a, i) := k$ iff $r(a, i) = \perp$ for all $a \in \mathcal{A}$ and $i \in \mathcal{I}$. To determine a ranking based on those labels, we define

$$\text{score}(a) := \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} w_{a,i} \cdot (\gamma(a, i) - 1) \quad , \tag{1}$$

with $w_{a,i} := 2$ (PAR-*2* penalty factor) if $\gamma(a, i) = k$ else $w_{a,i} := 1$. To be applicable to $\hat{a}$, we extend *score* by replacing $\gamma(a, i)$ by the model's predictions $f\colon (\mathcal{A} \cup \{\hat{a}\}) \times \mathcal{I} \to \{1, \ldots, k\}$.

Turning a regression into a classification problem, discretization is a useful tool. However, it does not provide much value if the resulting cluster labels are not predictive of the PAR-2 ranking we seek to predict. In our work, we use a hierarchical-clustering approach with a single-link criterion for $k = 3$. For the SAT Competition 2022 Anniversary Track instances [2], this ranking approach is capable of correctly deciding for almost all pairs of solvers which one is faster (about 97.45 %; $\sigma = 3.68$ %). Moreover, it decides all solvers correctly if the difference in the cluster-label-ranking scores between two solvers is at least 0.11. The Spearman correlation [6] of cluster-label and PAR-2 ranking is about 0.988 (in comparison to a perfect value of 1). This shows that monotonicity and score distances between ranks align quite well between both ranking methods.

That discretized runtimes are sufficient for distinguishing solvers can also be shown by comparing the fraction of solver pairs whose runtimes are significantly different. Within the SAT Competition 2022 Anniversary Track, about 89.95 % of solver pairs are significantly different (according to a Wilcoxon-signed-rank test with $\alpha = 0.05$). While only time-out labels (solver times out or not on a particular instance) are sufficient to significantly distinguish about 87.04 % of solver pairs, discrete-runtime labels are sufficient to distinguish about 87.83 % of solver pairs. This means that we do not lose much discriminatory power using discrete runtime labels. The predictiveness of the PAR-2 score, the labels' discriminatory power, as well as that it enables us to use classification, show the clear advantage of runtime discretization.

3.2 Instance Sampling

Our instance-sampling strategies (also refer to Alg. 1; line 4) can be divided into two categories. First, we present possible baseline approaches including existing

work on instance-based selection strategies [3,12,25]. Second, we show promising machine-learning-based sampling strategies that make use of the model's label-prediction function f and are inspired by existing work within the realms of active learning [35].

Baselines and existing approaches. Apart from obvious baseline instance-selection strategies, i.e., random sampling and neighborhood-aware sampling (randomly choosing among the instances with the highest amount of non-sampled k neighbors), we also implement existing instance-selection strategies.

Bossek and Wagner [3] introduce a custom ranking-score metric incorporating the number of *good* pairs and *bad* pairs. A good solver pair, regarding a particular instance, are solvers that perform according to their global ranking. Otherwise, it is a *bad* pair. Based on this classification, they assign a custom score to each instance. By selecting instances with the highest score, they try to favor instances that are predictive of the final ranking.

Gent et. al. [12] make use of a concept known from racing for automated model selection. Given a particular instance and a parameter ρ, a solver is ρ-dominated if it is at least ρ times worse than the best solver on this problem instance. By selecting instances with the highest amount of ρ-dominated solvers, they hope to sample instances that provide the best discriminatory power to distinguish the runtimes of a new contestant.

Finally, we also implement some of the selection approaches described by Matricon et.al. [25]. Besides uniformly random sampling and the aforementioned discrimination-based method by Gent et. al. [12], they also describe a variance-based selection approach. The variance-based strategy selects instances with the highest runtime variance per average runtime. We also implement a logarithmized version of this, i.e., the highest log-runtime variance per average log runtime.

Active-learning-based instance selection. For conciseness, we only present two of our active-learning-based instance-selection approaches, which show the most promising results. We implement a model-uncertainty-based approach and a model-based information-gain sampling strategy.

These methods require the model's predictions to also include probabilities for the k possible runtime labels, i.e.,

$$f' \colon (\mathcal{A} \cup \{\hat{a}\}) \times \mathcal{I} \to [0,1]^k \quad . \tag{2}$$

The model-uncertainty-based approach simply selects the instance that is closest to the model's decision boundary and, of course, not already selected, i.e.,

$$\underset{i \in \mathcal{I} \setminus \tilde{\mathcal{I}}}{\arg\min} \left| \frac{1}{k} - \max_{n \in \{1,\dots,k\}} f'(\hat{a}, i)_n \right| \quad . \tag{3}$$

The model-based information-gain sampling strategy selects the instance with the highest expected entropy reduction. To be more specific, we select the

instance i that maximizes

$$\mathrm{IG}(i) := \mathrm{H}(i) - \sum_{n=1}^{k} f'(\hat{a}, i)_n \, \mathrm{H}(i, n) \quad , \tag{4}$$

whereby $\mathrm{H}(i)$ denotes the entropy of the runtime labels $\gamma(a, i)$ for all $a \in \mathcal{A}$ and $\mathrm{H}(i, n)$ denotes the entropy of the runtime labels including the runtime label of $\hat{a}$ which is n. The term $\mathrm{H}(i, n)$ is computed for every possible runtime label n.

3.3 Stopping Criteria

Similar to the instance-selection strategies, we can also divide stopping criteria into baseline and active-learning-based approaches. Also, refer to Alg. 1, line 3. While a fixed subset-size criterion, e.g., only sample 20 % of all available instances, is an example of the former, a stopping criterion based on a Wilcoxon signed-rank test or the convergence of the model's predictions is an example for the latter.

The Wilcoxon stopping criterion stops the iterative active-learning algorithm when we are on average certain enough that the predicted runtime labels are sufficiently different in comparison to the other solvers. To be more specific, we stop when

$$W_{\hat{a}} := \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \mathrm{wilcoxon}_{\alpha}(\gamma(a, \mathcal{I}) \, , f(\hat{a}, \mathcal{I})) \tag{5}$$

drops below a certain threshold. Thereby, *wilcoxon* denotes the p-value of a Wilcoxon signed-rank test of the runtime-label distributions and $\gamma(a, \mathcal{I})$ and $f(\hat{a}, \mathcal{I})$ are the natural set extensions on γ and f respectively. To improve the stability of this criterion, we use an exponential moving average

$$W_{\mathrm{exp}}^{(m)} := \beta W_{\hat{a}} + (1 - \beta) \, W_{\mathrm{exp}}^{(m-1)} \quad , \tag{6}$$

whereby m denotes the count of iterations within Alg. 1 and $W_{\mathrm{exp}}^{(0)} := 1$.

The model-convergence stopping criterion is less sophisticated in comparison. It simply stops the active-learning process (Alg. 1) if within the last l iterations the ranking that is induced by the model's predictions (Eq. 1) remains unchanged. Note that the concrete values of $\mathrm{score}(a)$ might still change. We are solemnly interested in the induced ranking in this case.

4 Experimental Design

Given all the previously presented instantiations for Alg. 1, this section briefly outlines our experimental design, including our optimization goal, evaluation framework, used data sets, hyper-parameter choices, as well as implementation details.

4.1 Optimization Goal

As already stated in the introductory section, this work addresses the *New-Solver Problem* (cf. Def. 1). By running Alg. 1, we obtain a prediction model $\mathcal{M}$ that provides us with a scoring function *score*. Also, it outputs the fraction of runtime that is needed to arrive at its conclusion. First and foremost, our goal is to provide the engineer of new SAT solvers with an accurate ranking. 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 by the given ranking (cf. Alg. 2). For now, we exclude the equality of solvers since data shows that the PAR-2 scores of all our solvers are different. So, possible ranking decisions may be, for example, solver a is better than b or b is better than a. The ranking accuracy is affected by whether this decision is correctly made. Matricon et. al. [25] also evaluate their approach by the fraction of solver pairs that is correctly decided. Second, we also have to optimize for runtime. 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 (cf. Alg. 2). Overall, we want to find an approach that maximizes

$$O_\delta := \delta O_{\text{acc}} + (1 - \delta)\,(1 - O_{\text{rt}}) \quad , \tag{7}$$

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 a O_{rt}-O_{acc}-diagram provides us with a Pareto front of the best approaches for different optimization-goal weightings.

4.2 Evaluation Framework

To evaluate a concrete instantiation of Alg. 1 (a concrete choice for all the subroutines), we perform cross-validation on our set of solvers. That means that each solver once plays the role of the new solver. Alg. 2 shows this. Note that the *new* solver in each iteration is excluded from the set of solvers $\mathcal{A}$ to avoid data leakage. Also, the runtime function r is restricted to $\mathcal{A} \times \mathcal{I}$.

4.3 Data

In our experiments, we work with the SAT Competition 2022 Anniversary Track instances [2]. The dataset consists of 5355 instances with respective runtime data. It has complete runtimes of 28 solvers. We use the GBD metadata database [20] to provide us with the aforementioned instance features, problem instances, and solver runtimes. Thereby, all our approaches make use of the 56 base features that are maintained in the *base* database of GBD [20]. They are inspired by the SATzilla features [42]. Those features comprise general instance-size features and graph-representation features among others. All features are numeric and fortunately free of any missing values. We drop 10 out of 56 features because of zero variance. For hyper-parameter tuning, we randomly sample 10 % of the

Algorithm 2: Evaluation framework

Input: A pool of instances $\mathcal{I}$, a set of solvers $\mathcal{S}$ with known runtimes $r\colon \mathcal{S} \times \mathcal{I} \to [0, \tau] \cup \{\perp\}$, and PAR-2 scores $par2\colon \mathcal{S} \to \mathbb{R}$ for each solver.

$res \leftarrow \emptyset$
for $\hat{a} \in \mathcal{S}$ **do**
  $\mathcal{A} \leftarrow \mathcal{S} \setminus \{\hat{a}\}$
  $\left(score, \tilde{\mathcal{I}}\right) \leftarrow \text{runALAlgorithm}\left(\mathcal{I}, \mathcal{A}, r|_{\mathcal{A}\times\mathcal{I}}, \hat{a}\right)$ *// Refer to Alg. 1*
  $O_{\text{acc}} \leftarrow \left(\sum_{a\in\mathcal{A}} [\text{sign}(score(a) - score(\hat{a})) = \text{sign}(par2(a) - par2(\hat{a}))]\right) / |\mathcal{A}|$ *// Iverson-bracket notation*
  $O_{\text{rt}} \leftarrow \left(\sum_{i\in\tilde{\mathcal{I}}} r(i, \hat{a})\right) / \left(\sum_{i\in\mathcal{I}} r(i, \hat{a})\right)$
  $res \leftarrow res \cup \{(O_{\text{acc}}, O_{\text{rt}})\}$
$\left(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}}\right) \leftarrow \text{mean}(res)$

Output: Return the average ranking accuracy $\bar{O}_{\text{acc}}$ and the average fraction of runtime needed $\bar{O}_{\text{rt}}$.

complete set of 5355 instances with stratification regarding the instance's family. All instance families that are too *small*, i.e., 10 % of them corresponds to less than one instance, are put into one meta-family for stratification. This smaller dataset allows for a more extensive exploration of hyper-parameter space.

4.4 Hyper-parameters

Given Alg. 1, there are several possible instantiations for the three phases, i.e., *selection*, *stopping*, and *ranking*. Also, there are different choices for the runtime-label prediction model. Note that we are not considering all previously listed approaches since a grid search of all combinations would be infeasible. Rather, we filter approaches based on preliminary experimental results (cf. Sec. 5.1) and do the main end-to-end experiments only with a subset. The end-to-end experiment configurations are given below.

Ranking. Regarding *ranking* (cf. Sec. 3.1), we experiment with the following approaches, including our used hyper-parameter values:

- Observed PAR-2 ranking of already sampled instances
- Predicted ranking induced by runtime-label predictions
  - History size: consider the latest 1, 10, 20, 30, or 40 predictions within a voting approach for stability. Given a particular ranking decision, the latest x predictions vote on the winner.
  - Fallback threshold: if the difference of scores between two solvers drops below 0.01 , 0.05 , or 0.1 , use the partially observed PAR-2 ranking as a tie-breaker. As discussed in Sec. 3.1, ranking decisions are all correct if the difference in scores between two solvers exceeds a certain value.

Selection. For *selection* (cf. Sec. 3.2), we experiment with the following methods, including our used hyper-parameter values:

- Random sampling
- Model-based 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: prefer instances with the greatest uncertainty per average runtime (`True` or `False`).
- Model-based 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: prefer instances with the greatest information-gain per average runtime (`True` or `False`).
- Neighborhood-aware random sampling (preliminary experiments only); randomly chooses among the instances with the highest amount of non-sampled k neighbors
- Ranking-based sampling [3] (preliminary experiments only)
- Discrimination-based sampling [12] (preliminary experiments only)
- Variance-based sampling [25] (preliminary experiments only)

Stopping. For *stopping* decisions (cf. Sec. 3.3), we experiment with the following criteria, including our used hyper-parameter values:

- Fixed subset size of 10 % or 20 % of instances
- Ranking convergence 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 the last 1 % or 2 % of sampled instances.
- Wilcoxon 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 $\beta = 0.7$.

Runtime-label prediction model. As a runtime-label prediction model, we only make use of one method that has proven to be successful in preliminary experiments (cf. Sec. 5.1). Our model of choice is an ensemble, stacking a quadratic-discriminant analysis[3] onto a random forest model[4] using a decision tree[5] to weight its members. If not stated explicitly, all other model parameters default to the presets of *scikit-learn*[6] [33].

[3] With a regularization parameter of zero and a singular value threshold of zero
[4] With *entropy* splitting criterion and *balanced* class weights
[5] With *Gini* impurity splitting criterion; choosing the best split for each branching decision and maximum tree depth of 5 levels
[6] https://scikit-learn.org/stable/index.html; Version 1.0.2

4.5 Implementation Details

For reproducibility, our source code, all experiments, and data are available on GitHub[7]. Our code is implemented in PYTHON using *scikit-learn* [33] for making predictions and *gbd-tools* [20] for SAT-instance retrieval.

5 Evaluation

Now that we are all set, we want to, first, discuss preliminary results including the choice of the runtime-discretization method and the choice of the machine-learning predictor. Thereafter, we look at end-to-end results and instance-family importance.

5.1 Preliminary Results

An exhaustive grid search of all approaches outlined in Sec. 3 is infeasible. We are, therefore, narrowing down the search by filtering based on preliminary results. Since all relevant decisions, i.e., *selection*, *stopping*, and *ranking*, revolve around the choice of prediction model, we are first looking at the *Matthews Correlation Coefficient* (MCC) scores [14,26] that a particular prediction model achieves. MCC scores are great in dealing with class imbalances in contrast to conventional metrics like accuracy.

Table 2: Time-out prediction Matthews Correlation Coefficient performances. Results are obtained using cross-validation. We provide 64 % of ground-truth for training, use 16 % as a validation set, and 20 % as a test set (5-fold cross-validation).

Prediction Model	Avg. MCC (Std.)
Stacking ensemble (QDA, RF)	0.9527 (0.0292)
Quadratic-discriminant analysis (QDA)	0.9290 (0.0339)
Random forest (RF)	0.8530 (0.0479)
AdaBoost with decision trees	0.8384 (0.0444)
AdaBoost with logistic regression	0.8158 (0.0985)
Decision tree	0.8059 (0.0707)
Logistic regression	0.8052 (0.1018)
kNN classifier	0.7885 (0.1521)
MLP classifier	0.7760 (0.1408)
Support-vector classifier	0.7757 (0.2149)
Naive Bayes	0.7306 (0.1394)

To obtain the results in Tab. 2, we perform cross-validation among all solvers: we assume that all other solver runtimes are known and provide 64 % of ground-truth runtime labels of the target solver for training, use 16 % as a validation set

[7] https://github.com/mathefuchs/al-for-sat-solver-benchmarking

for hyper-parameter tuning, and 20 % as a test set (5-fold cross-validation). This is repeated for each solver once. Note that we perform cross-validation on two levels, i.e., among the solvers and the target solver runtimes. We report the average performances on the test sets. Since the differences in the MCC-performance distributions of the best and second-best approaches are statistically significant regarding a Wilcoxon signed-rank test with $\alpha = 5\,\%$, we only use the best approach for all further experiments as a prediction model.

As a runtime discretization technique, we use hierarchical clustering with $k = 3$ using a log-single-link criterion for all further experiments. Each non-time-out 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. As already mentioned in Sec. 3.1, this approach has good properties regarding the predictiveness of the PAR-2 score ranking and its discriminatory power. Also, preliminary experiments showed that our stacking ensemble performed particularly well for this kind of discretization method. Other clustering approaches that we have tried include hierarchical clustering with mean-, median- and complete-link criteria, as well as k-means and spectral clustering. Predicting runtime labels in a similar experiment as above produced an MCC score of on average 88.82 % ($\sigma = 4.75\,\%$; in comparison to 95.27 % for predicting time-outs).

5.2 Experimental Results

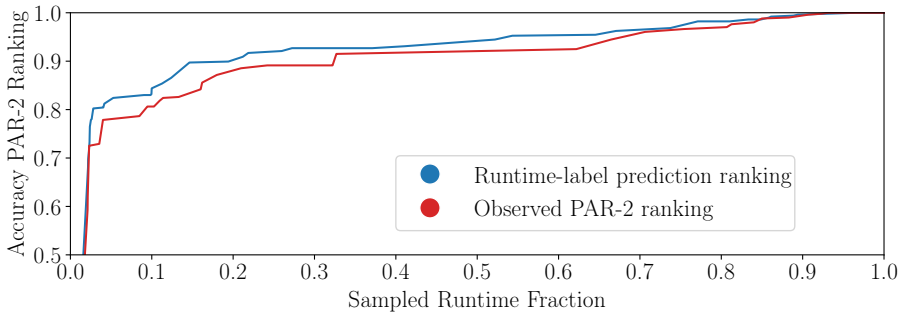
Tuning our algorithm. Our main end-to-end experiments follow the evaluation framework that has been introduced in Sec. 4.2. Fig. 2 shows the performance of the approaches listed in Sec. 4.4 on O_{rt}-O_{acc}-diagrams for the smaller hyper-parameter-tuning dataset. The plotted lines represent the Pareto front for each possible choice of ranking approaches (Fig. 2a), selection approaches (Fig. 2b), and stopping criteria (Fig. 2c). Colors indicate the different hyper-parameter instantiations. To be more specific, they represent the Pareto front of all configurations with the respective instantiations.

Regarding the ranking approach (Fig. 2a), using the ranking that is induced by our runtime-label predictions consistently performs better than the partially observed PAR-2 ranking for each possible value of δ. This 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.

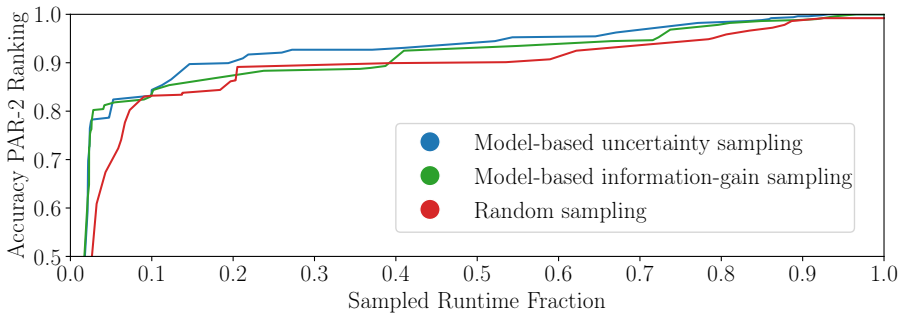
Regarding the selection approaches (Fig. 2b), the model-based uncertainty strategy performs best in most cases. However, the model-based information-gain sampling is beneficial if runtime is strongly favored (very small δ; runtime fraction less than 5 %).

Regarding the stopping criterion (Fig. 2c), the ranking convergence criterion performs the most consistently well. If PAR-2 accuracy is strongly favored (very high δ), the Wilcoxon stopping criterion performs better.

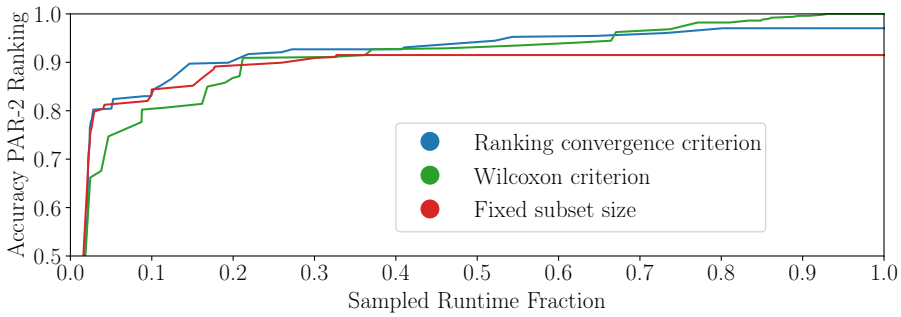
Tab. 3 also shows the performance of particular hyper-parameter choices for different values of δ. Thereby, we observe that for small δ (runtime more



(a) O_{rt}-O_{acc}-diagram showing Pareto fronts for configurations with the given ranking approaches. The lines show the best performances of approaches using runtime-label-prediction ranking and partially observed PAR-2 ranking respectively.



(b) O_{rt}-O_{acc}-diagram showing Pareto fronts for configurations with the given instance-selection approaches. The lines show the best performances of approaches using uncertainty-based, information-gain-based, and random instance selection respectively.



(c) O_{rt}-O_{acc}-diagram showing Pareto fronts for configurations with the given stopping criteria. The lines show the best performances of approaches using the ranking-convergence criterion, Wilcoxon criterion, and a fixed subset size respectively.

Fig. 2: O_{rt}-O_{acc}-diagrams showing the performance of different hyper-parameter instantiations. The x-axis shows the fraction of runtime in comparison to the time needed to evaluate a solver on all instances. The y-axis shows the accuracy of the predicted rank in comparison to the true PAR-2 rank. Each line entails the front of Pareto-optimal configurations with the respective instantiation.

Table 3: Pareto-optimal approaches for different choices of δ regarding our smaller hyper-parameter tuning dataset. Rows with no changes in comparison to their predecessor are hidden. Ranking approaches are either predicted cluster-label ranking (Pred.) or partially observed PAR-2 ranking (PAR-2). Selection approaches are either uncertainty-based selection (Uncert.) or information-gain-based selection (IG). Stopping criteria are either ranking convergence (Conv.) or Wilcoxon-based (Wilcoxon).

δ	Ranking	Selection	Stopping	Fraction Runtime (O_{rt})	Fraction Instances	O_{acc}	O_δ
0.00	PAR-2	Uncert.	Conv.	0.021	0.025	0.593	0.979
0.05	Pred.	Uncert.	Conv.	0.024	0.158	0.765	0.966
0.10	Pred.	Uncert.	Conv.	0.025	0.128	0.779	0.955
0.15	Pred.	IG	Conv.	0.028	0.112	0.802	0.946
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.50	Pred.	IG	Conv.	0.028	0.112	0.802	0.887
0.55	Pred.	Uncert.	Conv.	0.053	0.142	0.824	0.880
0.60	Pred.	Uncert.	Conv.	0.146	0.082	0.897	0.880
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.75	Pred.	Uncert.	Conv.	0.146	0.082	0.897	0.886
0.80	Pred.	Uncert.	Conv.	0.219	0.128	0.917	0.890
0.85	Pred.	Uncert.	Conv.	0.272	0.171	0.927	0.897
0.90	Pred.	Uncert.	Wilcoxon	0.894	0.888	0.996	0.907
0.95	Pred.	Uncert.	Wilcoxon	0.931	0.924	1.000	0.953
1.00	Pred.	Uncert.	Wilcoxon	0.989	0.981	1.000	1.000

important) our approaches preferably use easy instances to get a good sense of the PAR-2 ranking fast, e.g., row with $\delta = 0.15$. In contrast to that, harder instances are preferred for higher values of δ, i.e., the sampled fraction of runtime is greater than the fraction of instances. In our opinion, the row with $\delta = 0.85$ represents the best trade-off between runtime and PAR-2 accuracy.

Full dataset evaluation. Having selected the most promising hyper-parameters, we run our end-to-end active-learning experiments on the complete Anniversary Track dataset (5355 instances). The best-performing approach for $\delta = 0.85$ (since this was the most promising trade-off on the tuning set) uses runtime-label prediction ranking, model-based uncertainty sampling, and a ranking-convergence stopping criterion. Thereby, the sampled instances account for about 5.24 % of instances and about 10.35 % of runtime. Moreover, the PAR-2 accuracy is about 92.33 %, the average Spearman correlation between the PAR-2 ranking and the predicted ranking scores about 0.9572, and the weighted objective $O_\delta = 0.9193$.

Table 4: Top-10 most *important* families regarding the average occurrences and the fraction of selected occurrences regarding our AL approach

(a) Sorted by the average number of occurrences in the sample

Family	In Sample	In Dataset	Fraction (%)
planning	19.14	333	5.7
subgraph-isomorphism	16.27	175	9.3
crypto-graphy	13.11	311	4.2
pigeon-hole	7.55	151	5.0
fpga-routing	6.16	73	8.4
rbsat	5.93	113	5.2
quasigroup-completion	5.23	210	2.5
unknown	4.93	169	2.9
miter	4.84	199	2.4
bitvector	4.73	214	2.2
Remaining families	104.45	3407	3.1
Sum	192.34	5355	3.6

(b) Sorted by the fraction of sampled occurrences

Family	In Sample	In Dataset	Fraction (%)
circuit-multiplier	1.59	9	17.7
subset-cardinality	1.55	9	17.3
ssp-0	0.34	2	17.0
sliding-puzzle	1.52	11	13.8
knights-problem	2.25	17	13.2
genurq	1.55	12	12.9
stone	1.39	11	12.7
mosoi-289	3.75	31	12.1
anti-bandwidth	1.52	13	11.7
random	0.91	8	11.4
Remaining families	175.97	5232	3.4
Sum	192.34	5355	3.6

Instance-family importance. For all configurations of our approach that sample on average less than 10 % of all 5355 instances, we look at the particular instances and families that are selected by our active-learning approach. Thereby, the average benchmark set contains about 192.34 instances. Tab. 4a lists the ten most frequent families that are chosen. For example, *planning* instances account for on average 19.14 of the 192.34 benchmark instances. In contrast to that, Tab. 4b shows the families that are sampled the most often relative to their occurrences in the underlying dataset. There are, for example, only nine *circuit-multiplier* instances in the complete dataset. However, we sample on average 1.59 of them.

6 Conclusion

In this work, we have discussed possible solutions to the *New-Solver Problem*: Given a new solver, we want to find its ranking amidst its competitors. Our approaches provide accurate ranking predictions while only needing a fraction of the runtime resources that a full evaluation on all benchmarking instances would need. We use a runtime discretization technique as this enables us to transform the regression problem of directly predicting runtimes into the much

simpler notion of classification. We have shown that, albeit being more simple, the classification of discrete runtime labels produces good results. We have evaluated several ranking algorithms, instance-selection approaches, and stopping criteria within our sequential active-learning process. A model-uncertainty-based selection approach in combination with runtime-label-prediction ranking and a ranking-convergence stopping criterion showed the consistently best results. We also took a brief look at which instance families are the most prevalent when sampling instances.

6.1 Future Work

In future work, it may be of interest to compare further ranking algorithms, instance-selection approaches, and stopping criteria. Furthermore, it is possible to formulate the runtime discretization as an optimization problem. Given a dataset, optimization could select the discretization technique with the best discriminatory power rather than selecting one approach upfront by hand.

A major shortcoming is the lack of parallelizability. Our current approach selects instances one at a time. Running benchmarks on a computing cluster with n cores benefits from having a batch of n instances at a time. This is, however, not trivial since, on the one hand, a higher n leads to less *active learning* (because of bigger batch sizes) and, on the other hand, it is not clear how to synchronize the model update and instance selection without a barrier lock, which wastes a lot of runtime resources.

On a more general note, it would make sense to generalize this evaluation framework for arbitrary $\mathcal{NP}$-complete problems. Those problems share most of the relevant properties of SAT solving, i.e., there are established problem-instance features, a full benchmark run takes days, and creating new problem solvers traditionally requires expert knowledge to hand-select instances of interest.

References

1. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT Challenge 2012 solver competition. Artif. Intell. **223**, 120–155 (2015). https://doi.org/10.1016/j.artint.2015.01.002
2. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions. Department of Computer Science, University of Helsinki (2022), http://hdl.handle.net/10138/347211
3. Bossek, J., Wagner, M.: Generating instances with performance differences for more than just two algorithms. In: Proc. GECCO. pp. 1423–1432 (2021). https://doi.org/10.1145/3449726.3463165
4. Collautti, M., Malitsky, Y., Mehta, D., O'Sullivan, B.: SNNAP: solver-based nearest neighbor for algorithm portfolios. In: Proc. ECML PKDD. pp. 435–450 (2013). https://doi.org/10.1007/978-3-642-40994-3_28
5. Dang, N., Akgün, Ö., Espasa, J., Miguel, I., Nightingale, P.: A framework for generating informative benchmark instances. In: Proc. CP. pp. 18:1–18:18 (2022). https://doi.org/10.4230/LIPIcs.CP.2022.18

6. De Winter, J.C.F., Gosling, S.D., Potter, J.: 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 (2016). https://doi.org/10.1037/met0000079
7. Dehghani, M., Tay, Y., Gritsenko, A.A., Zhao, Z., Houlsby, N., Diaz, F., Metzler, D., Vinyals, O.: The benchmark lottery. arXiv:2107.07002 [cs.LG] (2021), https://arxiv.org/abs/2107.07002
8. Eggensperger, K., Lindauer, M., Hutter, F.: Pitfalls and best practices in algorithm configuration. J. Artif. Intell. Res. **64**, 861–893 (2019). https://doi.org/10.1613/jair.1.11420
9. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT Competition 2020. Artif. Intell. **301** (2021). https://doi.org/10.1016/j.artint.2021.103572
10. Garzón, I., Mesejo, P., Giráldez-Cru, J.: On the performance of deep generative models of realistic SAT instances. In: Proc. SAT. pp. 3:1–3:19 (2022). https://doi.org/10.4230/LIPIcs.SAT.2022.3
11. Gelder, A.V.: Careful ranking of multiple solvers with timeouts and ties. In: Proc. SAT. pp. 317–328 (2011). https://doi.org/10.1007/978-3-642-21581-0_25
12. Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, G.F., Nightingale, P.: Discriminating instance generation for automated constraint model selection. In: Proc. CP. pp. 356–365 (2014). https://doi.org/10.1007/978-3-319-10428-7_27
13. Golbandi, N., Koren, Y., Lempel, R.: Adaptive bootstrapping of recommender systems using decision trees. In: Proc. WSDM. pp. 595–604 (2011). https://doi.org/10.1145/1935826.1935910
14. Gorodkin, J.: Comparing two k-category assignments by a k-category correlation coefficient. Comput. Biol. Chem. **28**(5–6), 367–374 (2004). https://doi.org/10.1016/j.compbiolchem.2004.09.006
15. Guo, J., Pang, Z., Sun, W., Li, S., Chen, Y.: Redundancy removal adversarial active learning based on norm online uncertainty indicator. Comput. Intell. Neurosci. (2021). https://doi.org/10.1155/2021/4752568
16. Harpale, A., Yang, Y.: Personalized active learning for collaborative filtering. In: Proc. SIGIR. pp. 91–98 (2008). https://doi.org/10.1145/1390334.1390352
17. Hoos, H.H., Hutter, F., Leyton-Brown, K.: Automated configuration and selection of SAT solvers. In: Handbook of Satisfiability, chap. 12, pp. 481–507. IOS Press, 2 edn. (2021). https://doi.org/10.3233/FAIA200995
18. Hoos, H.H., Kaufmann, B., Schaub, T., Schneider, M.: Robust benchmark set selection for boolean constraint solvers. In: Proc. LION. pp. 138–152 (2013). https://doi.org/10.1007/978-3-642-44973-4_16
19. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. LION. pp. 507–523 (2011). https://doi.org/10.1007/978-3-642-25566-3_40
20. Iser, M., Sinz, C.: A problem meta-data library for research in SAT. In: Proc. PoS. pp. 144–152 (2018). https://doi.org/10.29007/gdbb
21. Kapoor, A., Grauman, K., Urtasun, R., Darrell, T.: Active learning with gaussian processes for object categorization. In: Proc. ICCV (2007). https://doi.org/10.1109/ICCV.2007.4408844
22. Koren, Y., Bell, R.M., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009). https://doi.org/10.1109/MC.2009.263
23. Körner, C., Wrobel, S.: Multi-class ensemble-based active learning. In: Proc. ECML. pp. 687–694 (2006). https://doi.org/10.1007/11871842_68

24. Manthey, N., Möhle, S.: Better evaluations by analyzing benchmark structure. In: Proc. PoS (2016), http://www.pragmaticsofsat.org/2016/reg/POS-16_paper_4.pdf
25. Matricon, T., Anastacio, M., Fijalkow, N., Simon, L., Hoos, H.H.: Statistical comparison of algorithm performance through instance selection. In: Proc. CP. pp. 43:1–43:21 (2021). https://doi.org/10.4230/LIPIcs.CP.2021.43
26. Matthews, B.W.: Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochim. Biophys. Acta - Protein Struct. **405**(2), 442–451 (1975). https://doi.org/10.1016/0005-2795(75)90109-9
27. Melville, P., Mooney, R.J.: Diverse ensembles for active learning. In: Proc. ICML (2004). https://doi.org/10.1145/1015330.1015385
28. Mısır, M.: Data sampling through collaborative filtering for algorithm selection. In: Proc. IEEE CEC. pp. 2494–2501 (2017). https://doi.org/10.1109/CEC.2017.7969608
29. Mısır, M.: Benchmark set reduction for cheap empirical algorithmic studies. In: Proc. IEEE CEC. pp. 871–877 (2021). https://doi.org/10.1109/CEC45853.2021.9505012
30. Mısır, M., Sebag, M.: ALORS: An algorithm recommender system. Artif. Intell. **244**, 291–314 (2017). https://doi.org/10.1016/j.artint.2016.12.001
31. Ngoko, Y., Cérin, C., Trystram, D.: Solving SAT in a distributed cloud: A portfolio approach. Int. J. Appl. Math. Comput. Sci. **29**(2), 261–274 (2019). https://doi.org/10.2478/amcs-2019-0019
32. Nießl, C., Herrmann, M., Wiedemann, C., Casalicchio, G., Boulesteix, A.: Over-optimism in benchmark studies and the multiplicity of design and analysis options when interpreting their results. WIREs Data Min. Knowl. Discov. **12**(2) (2022). https://doi.org/10.1002/widm.1441
33. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Édouard Duchesnay: Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. **12**(85), 2825–2830 (2011), http://jmlr.org/papers/v12/pedregosa11a.html
34. Rubens, N., Elahi, M., Sugiyama, M., Kaplan, D.: Active learning in recommender systems. In: Recommender Systems Handbook, chap. 24, pp. 809–846. Springer, 2 edn. (2015). https://doi.org/10.1007/978-1-4899-7637-6_24
35. Settles, B.: Active learning literature survey. Tech. rep., University of Wisconsin-Madison, Department of Computer Sciences (2009), http://digital.library.wisc.edu/1793/60660
36. Sinha, S., Ebrahimi, S., Darrell, T.: Variational adversarial active learning. In: Proc. ICCV. pp. 5971–5980 (2019). https://doi.org/10.1109/ICCV.2019.00607
37. Stützle, T., López-Ibáñez, M., Pérez-Cáceres, L.: Automated algorithm configuration and design. In: Proc. GECCO. pp. 997–1019 (2022). https://doi.org/10.1145/3520304.3533663
38. Tong, S., Koller, D.: Support vector machine active learning with applications to text classification. J. Mach. Learn. Res. **2**, 45–66 (2001), http://jmlr.org/papers/v2/tong01a.html
39. Tran, T., Do, T., Reid, I.D., Carneiro, G.: Bayesian generative active deep learning. In: Proc. ICML. pp. 6295–6304 (2019), http://proceedings.mlr.press/v97/tran19a.html
40. Volpato, R., Song, G.: Active learning to optimise time-expensive algorithm selection. arXiv:1909.03261 [cs.LG] (2019), https://arxiv.org/abs/1909.03261

41. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008). https://doi.org/10.1613/jair.2490
42. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Features for SAT. Tech. rep., University of British Columbia (2012), https://www.cs.ubc.ca/labs/beta/Projects/SATzilla/Report_SAT_features.pdf