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. Benchmarking is one of the most important phases when developing algorithms. This also applies to solutions of the SAT (propositional satisfiability) problem. While the field of SAT solver benchmarking is well established, traditional benchmark selection approaches do not optimize for benchmark runtime. Benchmark selection chooses representative instances from a pool of instances such that they reliably discriminate SAT 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 running time and maximum rank prediction accuracy. We evaluated this approach using 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 %. In this paper, we also discuss the importance of instance families in the selection process. Overall, our tool provides a reliable way for SAT solver engineers to efficiently determine the performance of a new SAT solver.

Keywords: Propositional satisfiability · Benchmarking · Active learning

1 Introduction

One of the main phases of algorithm engineering is benchmarking. 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. Competitive SAT solvers are the result of extensive experimentation and a variety of ideas and considerations [8,2]. While the field of SAT solver benchmarking is well established, traditional benchmark selection approaches do not optimize for benchmark runtime. Instead, the primary goal of traditional approaches is to select a representative set of instances for ranking solvers with a scoring scheme [10,16]. SAT Competitions typically use the widely adopted 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 is able to predict the rank of a new solver with high accuracy in only a fraction of the time needed to evaluate the full benchmark. The problem we solve is specified in Definition 1.

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

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

Our approach satisfies several desirable criteria for benchmarking: It outputs a *ranking* to compare the new solver against the existing set of solvers. For this ranking, we do not even need to predict exact solver runtimes, which is trickier. We optimize the *runtime* that our strategy needs to arrive at its conclusion. We use problem-instance and runtime *features*. Moreover, we select instances *non-randomly* and *incrementally*. In particular, we take runtime information from already done experiments into account when choosing the next. By doing so, we can control the properties of the benchmarking approach such as its required runtime. Rather than solely outputting a binary classification, i.e, the new solver is worse than an existing solver, we provide a *scoring* function that shows by which margin a solver is worse and how similar it is to existing solvers. 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]. Since a new solver idea is rarely best on the first try, 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.

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

All our source code[1] and and data[2] are available on GitHub.

2 Related Work

Benchmarking is not only of high interest in many fields but also an active research area on its own. Recent studies show that benchmark selection is a difficult endeavor for multiple reasons. Biased benchmarks can easily lead to fallacious interpretations [7]. Also, benchmark selection has many movable parts, e.g., performance measures, aggregation, and handling of missing values. Questionable research practices might modify these elements a-posteriori to fit expectations, leading to biased results [27]. In the following, we discuss related work from the areas of static benchmark selection, algorithm configuration, incremental benchmark selection, and active learning.

[1] temporary, anonymized version for review: xxx

[2] temporary, anonymized version for review: xxx

Feature	Hoos [16]	SMAC [17]	Matricon [21]	Our approach
Ranking	✓	✗	✓	✓
Runtime Minimization	✗	✓	✓	✓
Incremental/Non-Random	✗	✗	✓	✓
Scoring	✓	✗	✗	✓
Scalability	✓	✓	✗	✓

Table 1: Support for desired criteria from Section 1. The table compares a static benchmark-selection approach [16], an algorithm configuration system [17], an existing active-learning approach [21], as well as our approach.

Static Benchmark Selection. Benchmark selection is an important issue for competitions, e.g., the SAT Competition. In such competitions, the organizers define the rules for composing the corresponding benchmarks. Balint et al. provide an overview of benchmark-selection criteria in different solver competitions [1]. 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 [24].

Hoos et al. discuss which properties are most desirable when selecting SAT benchmark instances [16]. The selection strategy presented is static, i.e., it does not depend on particular solvers to distinguish. 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 [35,36]. Their approach *ranks* solvers, is *feature-based*, and *scalable* in the sense that the evaluation time scales linearly with more instances and solvers. Also, it allows *scoring* and thereby comparing solvers. However, the approach does not optimize for benchmark *runtime* and selects instances *randomly*, apart from constraints on the instance hardness and feature distance.

Algorithm Configuration. Further related work can be found within the field of algorithm configuration [15,32], e.g., the configuration system SMAC [17]. There, 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 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. 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. While this approach considers the configuration's *runtime*, is *feature-based*, and *scales* by having a fixed time budget, it does not satisfy the other three criteria listed in Table 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.

Second, while using a model-based selection strategy to sample configurations, instance sampling is done *randomly*, i.e., without building a model over instances. And third, only few configurations within their approach are *scored* since algorithm configuration seeks to find solemnly the best performing configuration.

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 is dependent 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. Most of the proposed strategies in [21] are instance-based, i.e., no model of the solver runtimes is built. Regarding our desired criteria of benchmark selection (Table 1), the aforementioned approach *ranks* solvers, optimizes for *runtime*, is *feature-based*, and uses *incremental non-random* sampling. However, the approach only compares solvers binarily rather than providing a *scoring*. Thus, it is not clear how similar two given solvers are or on which instances they behave similarly. Moreover, a major shortcoming is the lacking *scalability* with the number of solvers. Comparing only pairs of solvers, evaluating a new solver requires sampling several benchmarks. In contrast, our approach allows comparing a new solver against a set of existing solvers with one benchmark.

Active Learning (AL). The posed *New-Solver Problem* has stark similarities to the well-studied field of active learning AL within recommender systems, especially the *new-user problem* [29]. 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. In contrast to traditional passive machine-learning methods with given instance labels, active learning allows for selecting instances for which to acquire labels. AL algorithms can be categorized into *synthesis-based* [5,9,33] and *pool-based* approaches [12,14,19].While synthesis-based methods generate instances for labeling, pool-based methods rely on a fixed set of unlabeled instances from which to sample.

Recent synthesis-based methods within the field of SAT solving show how to generate problem instances with desired properties. This goal is, however, orthogonal to ours [5,9]. 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. Volpato and Guangyan use pool-based AL to learn an instance-specific algorithm selector [34]. 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.

3 Active Learning for SAT Solver Benchmarking

In this section, we present the details of our approach. Section 3.1 describes the general framework. Section 3.2 specifies the details of the underlying prediction model and describes how we may derive a solver ranking from that model. We discuss possible criteria for sampling instances in Section 3.3. Section 3.4 concludes with possible stopping conditions.

3.1 Incremental Benchmarking Framework

Our framework to score a new solver $\hat{a}$ follows a three-step procedure, outlined in Algorithm 1: While the stopping condition is not met (line 1), we select *one* SAT instance from the pool of instances with the help of a prediction model $\mathcal{M}$ (line 2) and evaluate the new solver $\hat{a}$ on this instance (line 3). We then use the acquired result to update the prediction model $\mathcal{M}$ (line 4). When the stopping criterion is met, we return the predicted score of the new solver $\hat{a}$ (line 5).

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 [31,33].

Algorithm 1: Incremental Benchmarking Framework

`Input:` Model $\mathcal{M}$, Solver $\hat{a}$
`Output:` Score of $\hat{a}$

1 **while** not stop($\mathcal{M}$) **do** — *// cf. Section 3.4*
2   $s \leftarrow$ selectNextInstance($\mathcal{M}$) — *// cf. Section 3.3*
3   $t \leftarrow$ runExperiment($\hat{a}$, s) — *// Runs $\hat{a}$ on s with time-out τ*
4   updateModel($\mathcal{M}$, s, t) — *// cf. Section 3.2*
5 **return** predictScore($\mathcal{M}$) — *// cf. Section 3.2*

3.2 Solver Model

Let benchmark instances $\mathcal{I}$, SAT solvers $\mathcal{A}$, and runtimes $r : \mathcal{A} \times \mathcal{I} \rightarrow [0, \tau]$ be given. We denote the new solver to be ranked by $\hat{a} \notin A$ and define $\hat{A} := A \cup \{\hat{a}\}$ to denote all solvers including the new solver. Model $\mathcal{M}$ provides a prediction function $f : \hat{\mathcal{A}} \times \mathcal{I} \rightarrow \mathbb{R}$ that powers the decisions within our framework.

To be applicable to a new solver $\hat{a}$, whose runtimes are unknown, we extend *score* by replacing $\gamma(a, e)$ with the model's predictions $f : \hat{\mathcal{A}} \times \mathcal{I} \rightarrow \{1, \ldots, k\}$.

Runtime Transformation. We transform the runtimes into discrete runtime labels on a per-instance basis. For each instance $e \in \mathcal{I}$, we use a single-link hierarchical clustering algorithm to assign the runtimes in $\{r(a,e) \mid a \in A\}$ to one of the k clusters $C_1, \ldots, C_k$ such that the fastest runtimes for instance e are in cluster C_1 and the slowest are in cluster C_k. For instances with many timeouts, we have to manually fix the clustering results such that all timeouts τ are in cluster C_k. The runtime transformation function $\gamma_k : \mathcal{A} \times \mathcal{I} \to \{1, \ldots, k\}$ is then specified as follows:

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

Given an instance $e \in \mathcal{I}$, a solver $a \in A$ belongs to the $\gamma_k(a,e)$-fastest solvers on instance e. In preliminary experiments, we achieved higher accuracy predicting such discrete runtime labels than predicting raw runtimes. The mean squared error of raw runtime regression was within the same magnitude as the values to be predicted. Research on portfolio solvers has also shown that directly predicting runtimes without any transformation does not work well in practice [4,26].

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

However, we need to ensure that discretized labels still discriminate solvers and align with the actual PAR-2 ranking. We analyzed the ranking induced by s_3 in preliminary experiments with the SAT Competition 2022 Anniversary Track [2]. According to a Wilcoxon-signed-rank test with $\alpha = 0.05$, 87.83 % of solver pairs have significantly different scores after discretization, only a slight drop compared to 89.95 % before discretization. Further, our ranking approach is capable of correctly deciding for almost all (about 97.45 %; $\sigma = 3.68$ %) solver pairs which solver is faster. In particular, 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]. All these results show that discretized runtimes are suitable for our framework.

3.3 Instance Sampling

Selecting an instance based on the model is a core functionality of our framework (cf. Algorithm 1, Line 2). In this section, we introduce our sampling strategies, which make use of the model's label-prediction function f and are inspired by existing work within the realms of active learning [30]. We implement a model-uncertainty and an information-gain sampling strategy. These methods require the model's predictions to also include probabilities for the k possible runtime labels. Let $f' : \hat{\mathcal{A}} \times \mathcal{I} \to [0,1]^k$ denote this modified prediction function.

Model Uncertainty Sampling. The model-uncertainty sampling strategy simply selects the instance that is closest to the model's decision boundary.

$$\operatorname*{arg\,min}_{e \in \mathcal{I} \setminus \tilde{\mathcal{I}}} \left| \frac{1}{k} - \max_{n \in \{1,\ldots,k\}} f'(\hat{a}, e)_n \right|$$

Information Gain Sampling. The information-gain sampling strategy selects the instance with the highest expected entropy reduction regarding the runtime labels of the instance. To be more specific, we select the instance e 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 $n \in \{1, \ldots, k\}$.

3.4 Stopping Criteria

In this section, we present two dynamic stopping criteria, the Wilcoxon and the Model Uncertainty Stopping Criterion (cf. Algorithm 1, Line 1).

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

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

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 certain threshold:

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

Model Convergence Stopping Criterion The model-convergence stopping criterion is less sophisticated in comparison. It stops the active-learning process if the ranking induced by the model's predictions (Equation 1) remains unchanged within the last l iterations. Note that the concrete values of $s_k(\hat{a})$ 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 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. Definition 1). We create a prediction model $\mathcal{M}$ as described in Section 3.2 that provides us with a scoring function s_k. 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. Algorithm 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. 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. Algorithm 2). 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 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 Algorithm 1 (a concrete choice for all the sub-routines), we perform cross-validation on our set of solvers. That means that each solver once plays the role of the new solver. Algorithm 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 [18] to provide us with the aforementioned instance features, problem instances, and

Algorithm 2: Evaluation Framework

`Input:` Solvers $\mathcal{A}$, Instaces $\mathcal{I}$, Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$
`Output:` Average ranking accuracy $\bar{O}_{\text{acc}}$, Average fraction of runtime $\bar{O}_{\text{rt}}$

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

solver runtimes. Thereby, all our approaches make use of the 56 base features that are maintained in the *base* database of GBD [18]. They are inspired by the SATzilla features [36]. 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 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 Algorithm 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. Section 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. Section 3.2), 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. 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 Section 3.2, ranking decisions are all correct if the difference in scores between two solvers exceeds a certain value.

Selection. For *selection* (cf. Section 3.3), 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 (over all solvers) 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 [11] (preliminary experiments only)
- Variance-based sampling [21] (preliminary experiments only)

Stopping. For *stopping* decisions (cf. Section 3.4), 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 runtime-label prediction model, we only make use of one fixed model since an exhaustive grid search would be infeasible (cf. Section 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] [28].

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* [28] for making predictions and *gbd-tools* [18] for SAT-instance retrieval.

5 Evaluation

Before evaluating our approach end-to-end as described in Section 4, we discuss the choice of the runtime-discretization method and the machine-learning predictor. Thereafter, we look at the results of our active-learning approach. Selection decisions of the aforementioned approach also reveal the importance of different problem-instance families to our model. Instance families comprise instances that are derived from the same application domain, e.g., planning, cryptography, etc., and are a useful tool for both analyzing solver performance and portfolios.

5.1 Runtime Prediction

An exhaustive grid search of all active-learning approaches outlined in Section 3 and different runtime-prediction models is infeasible. Therefore, we use the runtime-prediction model that produced the best results for all further experiments. We compare runtime predictors by looking at the *Matthews Correlation Coefficient* (MCC) scores [13,22] of the classification task. MCC scores are great in dealing with class imbalances in contrast to conventional metrics like accuracy.

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 for hyper-parameter tuning of the runtime predictor, 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.

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

ML classification models that we have tried include random forests, decision trees, quadratic-discriminant analysis, AdaBoost, logistic regression, k-nearest neighbor, multi-layer perceptrons, support vector machines, and naive Bayes. A stacking ensemble consisting of quadratic-discriminant analysis and random forests performed best. It produces an MCC score of 0.9527 for time-out prediction and 0.8882 for runtime-label prediction. The results are significantly better than any other model that we have tried regarding a Wilcoxon signed-rank test with $\alpha = 5\,\%$.

For runtime discretization, 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 Section 3.2, this approach has good properties regarding the predictiveness of the PAR-2 score ranking and its discriminatory power. Also, 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.

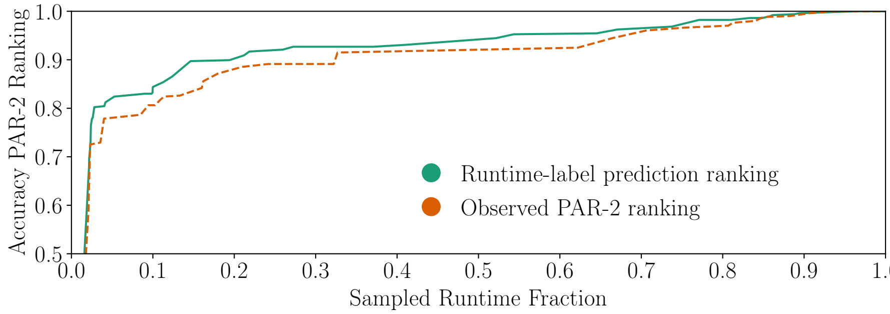
5.2 Experimental Results

Tuning our algorithm. Our main end-to-end experiments follow the evaluation framework that has been introduced in Section 4.2. Fig. 1 shows the performance of the approaches listed in Section 4.4 on O_{rt}-O_{acc}-diagrams for the smaller hyper-parameter-tuning dataset. Evaluating a particular configuration with Algorithm 2, returns a point (O_{rt}, O_{acc}). The plotted lines represent the best performing configurations (convex hull) per top-level hyper-parameter choice, i.e., per ranking approach (Fig. 1a), per selection approach (Fig. 1b), and per stopping criterion (Fig. 1c).

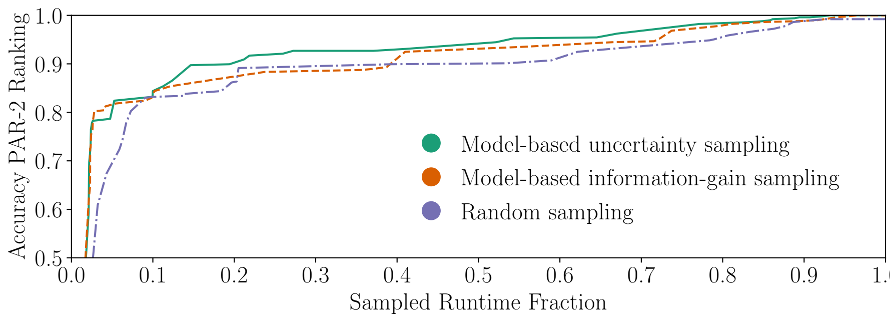
Regarding ranking approaches (Fig. 1a), using the s_3-induced ranking consistently outperforms the partially observed PAR-2 ranking for each possible value of δ. This is expected since selection decisions are not random. For example, we might sample more instances of one family if it benefits discrimination of solvers. While the partially observed PAR-2 score is skewed, the prediction model can account for this.

Regarding the selection approaches (Fig. 1b), 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 $5\,\%$). Random sampling is outperformed by the model-based uncertainty strategy in all cases showing the benefit of active learning. By actively choosing the next experiment to run based on a model, we are more accurate in predicting the PAR-2 ranking of the solvers on the underlying dataset.

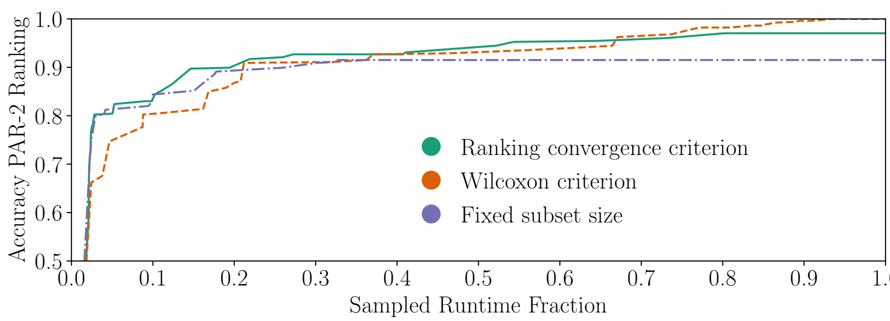
Regarding the stopping criterion (Fig. 1c), the ranking convergence criterion performs the most consistently well. If accuracy is strongly favored (very high δ), the Wilcoxon stopping criterion performs better.



(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. 1: 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.

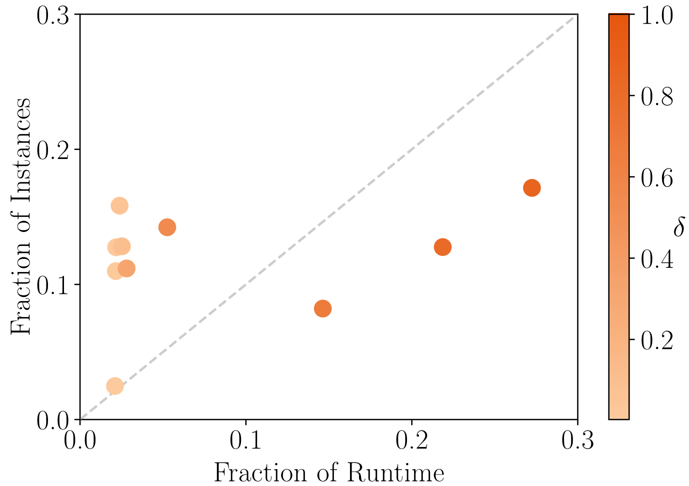


Fig. 2: Scatter plot showing different instantiations of our active-learning strategy for different δ on our smaller hyper-parameter-tuning dataset. The x-axis shows the fraction of runtime O_{rt} that a particular instantiation needs. The y-axis shows the fraction of instances that a particular instantiation selects to arrive at its conclusion. The color indicates the weighting between different optimization goals $\delta \in [0, 1]$. A small δ refers to favoring runtime over accuracy and a big δ refers to favoring accuracy over runtime.

Figure 2 also shows an interesting difference between different optimization goals. It shows how different configurations select instances with different difficulties. If it is, on the one hand, desired to get a *rough* estimate of a solver's performance fast (low δ), corresponding approaches favor selecting many *easy* problem instances (fraction of instances bigger than the fraction of runtime within $\tilde{\mathcal{I}}$). By having many observations, it is easier to build a model. If it is, on the other hand, desired to get a *good* estimate of a solver's performance in a moderate amount of time (high δ), corresponding approaches favor selecting fewer *difficult* problem instances (fraction of instances smaller than the fraction of runtime within $\tilde{\mathcal{I}}$).

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 $\delta = 0.85$ uses runtime-label prediction ranking, model-based uncertainty sampling, and a ranking-convergence stopping criterion. It can predict a new solver's PAR-2 ranking with about 92.33 % accuracy (O_{acc}), while only needing 10.35 % of the runtime that would be needed to evaluate the solver on the full dataset (O_{rt}). This trade-off provides a better way of prototyping and evaluating new SAT solvers rather than evaluating them on a manually picked subset based on domain knowledge. The sampled instances account for about 5.24 % of instances indicating the preference for selecting more difficult instances as already discussed in the previous section. The average Spearman correlation between the PAR-2 ranking and the predicted ranking scores is about 0.9572 indicating that the differences in PAR-2 scores are quite similar to s_3 scores. Because of that, the runtime-prediction model helps determine on what problem instances solvers perform similarly or are better/worse.

5.3 Instance-Family Importance

Selection decisions of our approach also reveal the importance of different problem-instance families to our model. Instance families comprise instances that are derived from the same application domain, e.g., planning, cryptography, etc., and are a useful tool for both analyzing solver performance and portfolios.

Figure 3 shows a scatter plot of problem-instance families regarding their occurrence within the dataset and the average occurrence in 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 benchmark set [2]. While most families appear by the same fraction in the dataset and in active-learning-selected benchmarks, there are a few outliers that need some further discussion. Problem instances of the families *fpga*, *quasigroup-completion*, and *planning* are especially helpful to our model in distinguishing solvers. Instances of these families are selected over-proportionally in comparison to their fraction in the dataset. In contrast to that, instances of the biggest family, i.e., *hardware-verification*, roughly appear with the same fraction in the dataset and in active-learning-selected benchmarks. Finally, instances of the family *cryptography* are less important in distinguishing solvers than their huge weight in the dataset suggests. A possible explanation is that *cryptography* instances are too similar, which means that a smaller fraction of them is sufficient to determine a solver's performance on all 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.

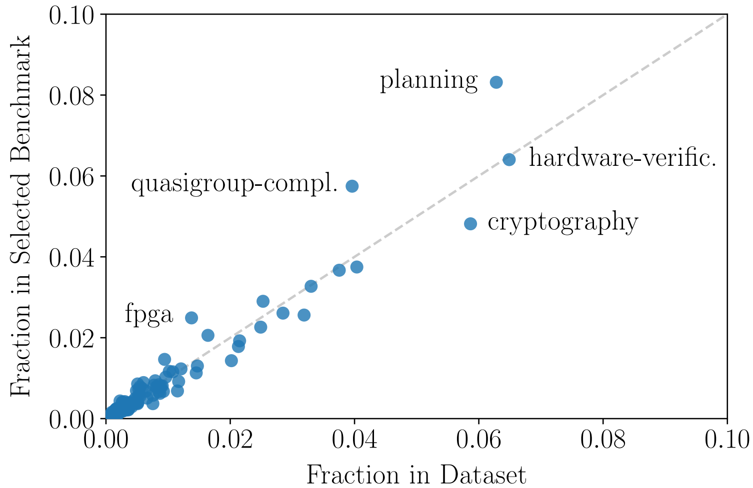


Fig. 3: Scatter plot showing the *importance* of different instance families to our model. The x-axis shows the fraction of instances within a particular family in the complete SAT Competition 2022 Anniversary Track benchmark set [2]. The y-axis shows the average fraction of instances within a particular family that is selected by our active-learning strategies. The dashed line represents families that are represented with the same fraction in, both, our dataset and the incremental benchmark sets $\tilde{\mathcal{I}}$ that are chosen by our approach.

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. 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
9. 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
10. 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
11. 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
12. 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
13. 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
14. Harpale, A., Yang, Y.: Personalized active learning for collaborative filtering. In: Proc. SIGIR. pp. 91–98 (2008). https://doi.org/10.1145/1390334.1390352
15. 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
16. 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

17. 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
18. 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
19. 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
20. 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
21. 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
22. 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
23. 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
24. 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
25. 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
26. 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
27. 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
28. 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
29. 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
30. Settles, B.: Active learning literature survey. Tech. rep., University of Wisconsin-Madison, Department of Computer Sciences (2009), http://digital.library.wisc.edu/1793/60660
31. 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
32. 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
33. 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

34. Volpato, R., Song, G.: Active learning to optimise time-expensive algorithm selection. arXiv:1909.03261 [cs.LG] (2019), https://arxiv.org/abs/1909.03261
35. 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
36. 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