Noname manuscript No.
(will be inserted by the editor)

Algorithm Selection for SMT

MachSMT: Machine Learning Driven Algorithm Selection for SMT Solvers

Joseph Scott · **Aina Niemetz** · **Mathias Preiner** · **Saeed Nejati** · **Vijay Ganesh**

Received: date / Accepted: date

Abstract This paper presents MachSMT, an algorithm selection tool for Satisfiability Modulo Theories (SMT) solvers. MachSMT supports the entirety of the SMT-LIB language and standardized SMT-LIB theories, and is easy to extend with support for new theories. MachSMT deploys machine learning (ML) methods to construct both empirical hardness models (EHMs) and pairwise ranking comparators (PWCs) over state-of-the-art SMT solvers. Given an input formula in SMT-LIB format, MachSMT leverages these learnt models to output a ranking of solvers based on predicted runtimes.

We provide an extensive empirical evaluation of MachSMT to demonstrate the efficiency and efficacy of MachSMT over three broad usage scenarios on theories and theory combinations of practical relevance (e.g., bit-vectors, (non-)linear integer and real arithmetic, arrays, and floating-point arithmetic). First, we deploy MachSMT on state-of-the-art solvers in SMT-COMP 2019 and 2020. We observe MachSMT frequently improves on the best performing solvers in the competition, winning 57 divisions outright, with up to a 99.4% improvement in `PAR-2` score. Second, we evaluate MachSMT to select configurations from a single underlying solver. We observe that MachSMT solves 898 more benchmarks and up to a 93.4% improvement in `PAR-2` score across 23 configurations of the SMT solver cvc5. Last, we evaluate MachSMT on domain-specific problems, namely network verification with simple domain-specific features, and observe an improvement of 77.3% in `PAR-2` score.

Keywords SMT Solvers · Machine Learning · Algorithm Selection

This work was supported in part by DARPA (award no. FA8650-18-2-7861) and ONR (award no. N68335-17-C-0558).

Joseph Scott, Saeed Nejati, and Vijay Ganesh
200 University Ave W, Waterloo, Ontario, Canada
E-mail: {joseph.scott, snejati, vijay.ganesh}@uwaterloo.ca

Aina Niemetz, Mathias Preiner
450 Jane Stanford Way, Stanford, CA 94305, USA
E-mail: {niemetz, preiner}@cs.stanford.edu

1 Introduction

Satisfiability Modulo Theories (SMT) solvers are tools to decide the satisfiability of formulas over first-order theories, such as the theories of bit-vectors, floating-point arithmetic, integers, reals, strings, arrays, and their combinations. Prominent examples of SMT solvers are Bitwuzla [59], Boolector [60], CVC4 [13], cvc5 [8], MathSAT [23], SMTInterpol [22], STP [29],Yices [25], and Z3 [13, 57]. In recent years, SMT solvers have had a revolutionary impact on applications in software engineering (broadly construed), such as software testing [20, 63] and verification [18, 28, 33, 50], as well as in sub-fields of AI [36, 44, 70]. This impact is a driver for an insatiable demand for evermore efficient solvers, not only to scale to larger instances obtained from existing applications (e.g., automatic bug-finding in commercial software [5, 31]), but also to solve problems from new application domains (e.g., verification and synthesis of cryptographic primitives [16]).

Motivation for Algorithm Selection for SMT Solvers. In response to this high demand, the SMT community has developed a plethora of solver heuristics and configurations. For example, in the 2019 edition [37] of the annual SMT-COMP competition [82], more than 50 solvers and their configurations were submitted. Many of these solvers implement very different algorithms to tackle the satisfiability problem for (a combination of) first-order theories, with significantly

varying performance profiles. For example, for the quantifier-free theory of floating-point arithmetic (QF_FP), there exist several substantially different decision procedures, e.g., word-blasting [19], abstract CDCL [17], inter-reduction methods [72], and reduction to global optimization [14,27]. In this specific setting of floating-point arithmetic, input instances may be derived from a variety of applications, such as software verification or analysis of machine learning (ML) models [74]. A very natural question arises in such a scenario: which solver or configuration is best for a given input instance?

Another well-known issue with many SMT solvers (even state-of-the-art ones) is that users may not know a priori which formula features or encoding would make an instance easy to solve. This can be very frustrating for users as they have to try a large number of different encodings with different solvers in order to find a combination that works best for their specific scenario, which may result in a combinatorial explosion. Users have also noted that as their applications change, what was once a great solver configuration in an earlier setting is suddenly underperforming in the newer one. One possible approach to address this problem is to use a portfolio of solvers, which has been successful in the context of SAT solvers. Unfortunately, given the plethora of SMT solvers and solver configurations (cvc5 [8] alone utilizes 23 different configurations in a sequential portfolio setting for quantified logics), such an approach quickly becomes infeasible.

One way to address the above-mentioned problems is to use an automated algorithm selection tool that can automatically and accurately predict the best algorithm from a given set of algorithms for a specific input. Such a tool selects the best SMT solver from a set of solvers for a given SMT formula. Note that in the following, in the context of algorithm selection, we will use *algorithm*, *solver configuration* and *solver* interchangeably when understood from the context.

Brief Overview of MachSMT. In this paper, we introduce MachSMT, a machine learning-based algorithm selection tool for SMT solvers. MachSMT supports the entirety of the SMT-LIB language [11] and standardized SMT-LIB theories [10], and is easy to extend with support for new theories. MachSMT is written in Python and takes as input an instance for a specified logic (a combination of theories with or without quantifiers) of interest and outputs a ranking of solvers predicted to have the lowest runtime. Internally, MachSMT is a set of machine learnt models constructed by analyzing the runtimes of solver configurations on benchmarks with respect to a predefined set of features, as given in Table 1. These features include the frequency of occurrences of grammatical constructs (e.g., predicates, operators, bindings) and syntactical features that can have an influence on performance (e.g., quantifier nesting levels).

At its core, MachSMT utilizes two techniques for algorithm selection: empirical hardness models (EHMs) and pairwise ranking comparators (PWCs). In addition, it pipelines the set of features defined in Table 1 with feature preprocessing and Principal Component Analysis (PCA). By default, MachSMT uses scikit-learn [65] with Adaboosting for classification and regression models. It is, however, ML framework-agnostic with a highly abstract interface and a built-in PyTorch interface [64].

An EHM for a given solver is a mapping from an input instance to a predicted runtime of the solver on that input. At runtime, MachSMT queries all EHMs for all solvers that were considered during training on the given input and outputs a ranking of solvers based on their predicted runtimes (the top-ranked solver is predicted to solve the input problem the fastest). In contrast, a learnt PWC is a mapping that takes as input a pair of solvers and an input instance, and outputs a ranking over the input solvers based on which is predicted to have a lower runtime on the given input. During evaluation, MachSMT uses the learnt PWC as a comparator to rank the set of solvers on a given input.

While algorithm selection has been considered in the broad setting of satisfiability solvers (e.g., QBF solvers [67] and SAT solvers [86]) as well as certain specific SMT theories [7, 75, 83], we are not aware of previous work on algorithm selection aimed at the entirety of SMT-LIB [10] and its standardized theories. Our results demonstrate that the MachSMT algorithm selector is highly effective, as it outperforms the best performing solvers in the competition on the majority of tracks from the SMT-COMP in 2019 and 2020.

Perhaps the first algorithm selection tool in the context of logic solvers was SATZilla [86]. Since its introduction, SATZilla has had a tremendous impact on SAT solver research, winning multiple gold medals in previous SAT competitions. MachSMT differs in several significant aspects from SATZilla. In particular, SATZilla deploys a feature selection scheme to avoid the curse of dimensionality, while MachSMT leverages a learnt dimensionality reduction scheme, Principal Component Analysis (PCA). In fact, a feature selection scheme would not scale in the context of SMT solvers given the very large number of learnt models that are incorporated into MachSMT. We discuss additional differences between SATZilla and MachSMT at length in Section 8.

It goes without saying that MachSMT is only as powerful as the underlying solvers that it has access to. MachSMT is clearly not a replacement for any particular SMT solver, but rather a tool that enables users

to leverage the collective strength of the diverse set of algorithms and configurations implemented as part of these sophisticated solvers.

Contributions. In this paper, we make the following contributions:

1. **The MachSMT Algorithm Selection Tool**. We present MachSMT, an algorithm selection tool for the entirety of SMT-LIB [10] and its standardized theories. Our tool uses machine learning (ML) to construct EHMs and PWCs of solvers for algorithm selection and is designed to be easily tuned and extended by SMT solver users (see Section 3). All source code is available at [1].
2. **Analysis of MachSMT over SMT-COMP 2019 and 2020 Benchmarks and Solvers**. We perform an empirical analysis of MachSMT across all divisions from SMT-COMP 2019 and 2020. We observe that MachSMT improves on the best performing solvers in the competition in 57 divisions, with up to 99.4% improvement in performance for the QF_BVFPLRA SQ'20 division, and up to 89.0% for the QF_UFBV SQ'19 division (see Section 4).
3. **Analysis of MachSMT for cvc5 Configuration Selection**: We perform an empirical analysis of MachSMT to select from 23 different configurations of the cvc5 SMT solver [8] across multiple logics. We observe that MachSMT solves an additional 898 benchmarks over the competition version of cvc5 (Section 5).
4. **Analysis of MachSMT Leveraging Domain-Specific Knowledge**: We perform an empirical analysis of MachSMT from three solvers (Bitwuzla [59], cvc5 [8], and Z3 [57]) to solve benchmarks from a network verification application. We evaluate MachSMT with and without domain-specific features (see Section 6). We observe MachSMT to considerably improve on the best standalone solver, and even more significantly with domain-specific features.

This article is an extension of the work presented in [73], and expands on leveraging the learnt models for resource allocation (Section 3.5). It further includes two novel experiments: using MachSMT to improve the scheduling of the sequential portfolio configurations of the SMT-COMP'21 submission of cvc5 (Section 5), and leveraging domain-specific knowledge to boost predictions of MachSMT (Section 6).

The rest of this paper is structured as follows. Section 2 provides the necessary background, Section 3 gives a technical description of MachSMT, Section 4 gives an experimental evaluation of MachSMT over SMT-COMP 2019 and 2020, Section 5 gives an experimental evaluation of MachSMT for solver configuration selection, Section 6 gives an experimental evaluation of MachSMT leveraging domain-specific knowledge, Section 7 provides an analysis of the experimental results, Section 8 describes related work, and Section 9 concludes the paper.

2 Background

In this section, we provide some background on SMT, algorithm selection via EHMs and PWCs, and the machine learning methods we use.

2.1 A Brief Overview of Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) solvers determine if a first-order formula is satisfiable with respect to some combination of background theories. Such background theories include the theories of arrays, bit-vectors, floating-point arithmetic, integers, reals, and strings, which all have a wide range of applications in verification, program analysis, and testing. The SMT-LIB input format provides a standardized syntax and semantics for a multitude of first-order theories and their combinations (also called "logics") [12]. In this paper, we use the following acronyms (given in brackets), optionally in combination, to refer to such SMT-LIB logics: quantifier-free (QF), arrays (A), uninterpreted functions (UF), bit-vectors (BV), floating-point arithmetic (FP), algebraic datatypes (DT), strings (S), integer arithmetic (IA), real arithmetic (RA), or mixed integer real arithmetic (IRA). Further, arithmetic over integers and reals can be linear (e.g., LIA, LRA), non-linear (e.g., NIA, NRA), or difference logics (IDL, RDL). For combinations of acronyms, their order appears in the order given above. In addition, each theory defines various operators (e.g., `bvadd` for bit-vector addition), predicates (e.g., `=` for equality), sorts (e.g., `Int` for integer sort, `Float32` for single-precision floating-point sort), and generalized keywords (e.g., `assert`, `check-sat`). For a complete list, as well as more details on syntax and semantics, we refer to the SMT-LIB standard [12].

2.2 A Brief Overview of Algorithm Selection

The idea of algorithm selection was first proposed and formalized by Rice [68] in 1976. It is common knowledge that given a set of different algorithms and implementations for the same specification or problem, it is often the case that one of these implementations

may perform poorly on a given class of inputs while another might perform very well. This is especially true for problems believed to be computationally hard (e.g., NP-hard). The reasons for this phenomenon could be as diverse as the choice of data structures, fundamental differences between algorithms, or the fact that heuristics implemented as part of one algorithm can exploit the input problem structure or the underlying hardware better than others.

It is natural to want to exploit the diversity in algorithmic approaches to minimize the cumulative runtimes. In practice, users often deploy *greedy* algorithm selection—picking the best-observed algorithm based on empirical analysis and testing. However, greedy algorithm selection can be sub-optimal when the empirically best algorithm has deficiencies relative to other algorithms on certain families of inputs.

With the recent advances in AI and ML, researchers are beginning to leverage these new technologies to advance algorithm selection. In the context of ML-driven algorithm selection for satisfiability solvers, to the best of our knowledge, there exist two fundamental approaches: the use of Empirical Hardness Models (EHMs) [87] and Pairwise Ranking Comparators (PWCs) [89].

2.2.1 Algorithm Selection via Empirical Hardness Models (EHMs)

Let $\mathcal{I}$ be an input in the language of a set of algorithms $\mathcal{S}$ with a corresponding feature vector $\vec{x} \in \mathbb{R}^n$. For an algorithm $s \in \mathcal{S}$, an EHM is a learnt function $f_s : \mathbb{R}^n \to \mathbb{R}$ that predicts the runtime of s on $\mathcal{I}$. An EHM is constructed with an ML regression model trained on collected runtime data. The algorithm is then selected by computing the global minimum of f_s over $\vec{x}$ as follows.

$$\operatorname*{argmin}_{s \in \mathcal{S}} f_s(\vec{x})$$

2.2.2 Algorithm Selection via Pairwise Ranking Comparators (PWCs)

For each unique solver pair (s, s') for $s, s' \in \mathcal{S}$ and $s \neq s'$, we construct a learnt comparator $f_{(s,s')} : \mathbb{R}^n \to \{0, 1\}$. Given a solver input $\mathcal{I}$ with its feature vector $\vec{x} \in \mathbb{R}^n$, the comparator $f_{(s,s')}(\vec{x})$ returns 0 if solver s solves $\mathcal{I}$ faster than s', and 1 otherwise. We compute a ranking of algorithms as a map r over $\mathcal{S}$, where $r[s]$ is the ranking of algorithm s that represents "how many algorithms in $\mathcal{S}$ are faster than s on input $\mathcal{I}$" as follows.

$$r[s] = \sum_{s' \in S}^{s \neq s'} f_{(s,s')}(\vec{x})$$

The selected algorithm is then the minimum ranked algorithm, computed as the argmin of $r[s]$ as follows.

$$\operatorname*{argmin}_{s \in \mathcal{S}} r[s]$$

2.3 Supervised Learning, Adaptive Boosting, Curse of Dimensionality, and K-Fold Cross-Validation

Supervised learning is one of the most predominant areas of ML. It takes as input a data set of features X and labels Y, where each data point $\vec{x} \in X$ has a label $y \in Y$. A data point is a real-valued vector $\vec{x} \in \mathbb{R}^n$ describing a sample. The learning problem is said to be a *classification* problem if the labels $y \in Y$ come from a fixed and finite set of classes $\mathcal{C}$ (e.g., a set of algorithms). Alternatively, the learning problem is a *regression* problem if the labels are real-valued (e.g., runtimes). In this paper, we will consider three regression techniques: AdaBoost, Deep Neural Networks, and Ridge Regression which are used in Sections 4, 5, and 6 respectively.

One efficient and effective approach to supervised learning is *Adaptive Boosting (AdaBoost)*. AdaBoost is an *ensemble* approach to machine learning invented by Freund and Schapire et al. [26], which won the Gödel Prize in 2003. In ensemble learning, a set of learning algorithms (e.g., weak learners) are trained, and predictions are made diplomatically across the set [51]. We use an ensemble of 200 decision trees in the AdaBoost algorithm. For more, we refer to Drucker et al. [24]. Adaboosting is one of the most popular supervised learning algorithms due to its out-of-the-box effectiveness in creating non-linear models.

For linear models, one efficient and effective approach is *Ridge Regression*, in which a linear model is trained by optimizing over mean square error and includes a L2 regularization penalty.

Finally, deep neural networks (DNN) are an extremely effective, biologically inspired, supervised learning technique that have had significant impact in a plethora of different domains, with a large amount of literature [32, 49]. As used in this work, a DNN is composed of multiple layers of neurons, where each layer corresponds to a learnt linear transformation followed by a non-linear activation function.

While supervised learning has had tremendous impact in several research areas, there are pitfalls such as the *curse of dimensionality* (CoD). For example, consider the convex polytope P formed around the convex hull of X. The volume of P increases exponentially with the dimensionality of X, requiring an exponential amount of data points to avoid extreme sparsity in X. Sparsity in data sets is one of the leading causes

of poor performance in learnt models [34]. There exists a large amount of literature on managing the CoD. In this paper, we discuss *feature selection* and deploy *dimensionality reduction* solutions. In feature selection, a new data set X' is computed from X by selecting the subset of most performant features on a validation data set. Feature selection was utilized in the SATZilla algorithm selection tool for Boolean satisfiability [86].

Despite the success of feature selection in SATZilla, feature selection does have some flaws. First, there is a significant loss of information. In the case of SATZilla, a feature vector composed of more than a hundred values describing an input is reduced to just five values. Second, the total number of feature subsets is exponential in the number of features. While there exist techniques to reduce the time spent searching for high-performing subsets [45,84], in our experiments, we found feature selection to be the most computationally expensive component of the SATZilla framework.

When evaluating the performance of a supervised learning model, a training set is used to construct the learnt model, and a testing set is set aside to evaluate. However, this method alone can be prone to overfitting and selection bias [56,71]. Instead, it is common to use k-fold cross-validation to evaluate learnt models. In k-fold cross-validation, the data set is first randomly partitioned into k subsets. The learnt model is then trained on $k-1$ sets and evaluated on the set that was excluded from training. This training and evaluation process is repeated for every subset to obtain fair predictions.

2.4 Unsupervised Learning and Principal Component Analysis

In contrast to supervised learning, unsupervised learning is the study of detecting patterns in an unlabelled data set X. Applications of unsupervised learning include dimensionality reduction [81, 85], clustering [35, 91], and anomaly detection [2, 48].

Principal Component Analysis (PCA) is an unsupervised learning dimensionality reduction technique, which computes an orthogonal transformation of a data set X composed of points in $\mathbb{R}^n$ to a new data set X' composed of points in $\mathbb{R}^{n'}$ where $n' < n$. It is an incremental algorithm where a new component (or dimension) is computed in each iteration. On the first iteration, a hyperplane is fit around the data set X, and its corresponding spanning vector is the first element of the basis around the transformation onto X'. On each subsequent iteration, a new hyperplane is computed under the additional constraint of being orthogonal to its predecessors. This process is repeated until the desired number of iterations is achieved [38, 85].

3 An Overview of MachSMT

In this section, we provide an overview of the MachSMT tool. The architecture diagram of MachSMT is presented in Figure 1.

3.1 Features and Preprocessing

By default, MachSMT defines a feature vector with 196 entries (i.e., dimensions). This feature vector can optionally be extended with user-defined features (see Section 3.6). A complete description of each predefined feature is provided in Table 1. We use two strategies to mitigate taxing feature calculation times, which can severely impair algorithm selection solutions. First, all features are entirely syntactical properties of the input. This is a major difference between MachSMT and other algorithm selection solutions, such as SATZilla. Second, all features are calculated within a strict and user-adjustable time limit (default 10s). On a timeout, the feature value is recorded as -1.0.

By default, MachSMT performs three key preprocessing steps before constructing any learnt models over a given data set. The entirety of the ML pipeline is mutable, and MachSMT has an interface to connect to any major ML platform. We describe each subsequently. First, all feature values are scaled to zero mean and unit variance[1]. This data normalization technique is common in ML research and applications to improve model efficiency and numerical robustness.

The second step in the preprocessing pipeline is computing the polynomial interaction terms of degree two on the normalized feature vector. These polynomial features make interacting correlations of features explicit. These first two preprocessing steps are included in the SATZilla preprocessing pipeline [90].

As discussed in Section 2, ML in a high dimensional space is prone to the curse of dimensionality. While other algorithm selection tools (e.g., SATZilla) commonly implement feature selection solutions, we propose to use learnt dimensionality reduction, namely, PCA. As discussed above, feature selection can be a proactive solution to the curse of dimensionality but presents many challenges when applied to SMT. Internally, MachSMT manages up to more than a thousand learnt models (e.g., in the context of our SMT-COMP analysis in Section 4), and calculating optimal feature subsets for each one is infeasible.

The third and final preprocessing step is (optionally) applying PCA to the polynomial features. The

[1] $\frac{x-\mu}{\sigma}$, where x is a feature sample, μ is the mean across the specific feature on the training set, and σ is the deviation across the specific feature on the training set.

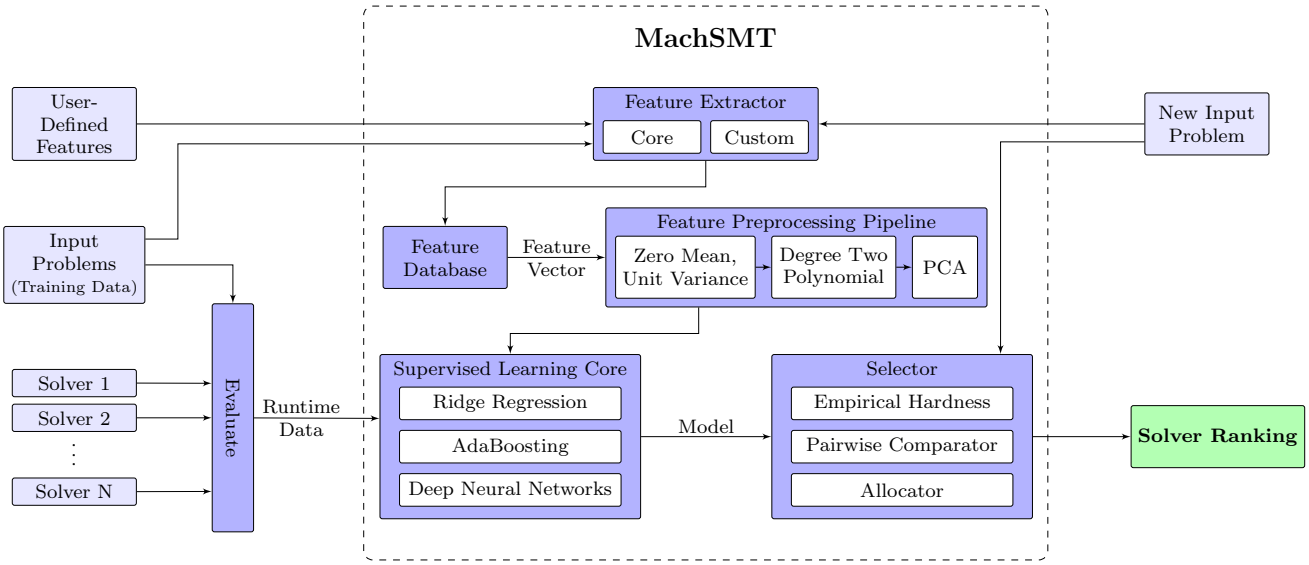


Fig. 1 *Architecture of MachSMT.*

final feature vector is composed of the first 35 principal components and is used to construct the learnt models with AdaBoost. This step can dramatically improve building times on larger data sets with many logics. Furthermore, several supervised learning techniques are more performant with PCA enabled. Concretely, MachSMT computes the 196+ features from the input, and the means and deviations are computed on each component in the feature vector across the provided data set. MachSMT then expands this vector by computing polynomial interaction features of degree 2 (i.e., products of all components), resulting in an expanded feature vector of nearly 40,000 dimensions. Finally, we reduce this to a vector space of just 35 dimensions with PCA, which aims to preserve as much of the original information as possible from this large vector space. The reduction to 35 dimensions is a hyperparameter of PCA and MachSMT, and was empirically determined.

3.2 Supervised Learning Core

Depending on the quality and type of data available, different machine learning techniques may have different tradeoffs, as well as development and production constraints. For example, linear ridge regression is a popular regression algorithm for algorithm selection [90] with relatively very fast training times and very interpretable models. It is, however, not always the most performant technique. On tabular data problems, decision trees and AdaBoosting increase the overall accuracy of the learned models at the cost of interpretability and training times. Furthermore, it may be beneficial to deploy deep learning techniques in the presence of large amounts of data or significant resources.

MachSMT is written in Python and aims to be ML framework and algorithm agnostic. By default, MachSMT leverages scikit-learn [65] and numpy [39]. However, it is easy to extend MachSMT to support any ML model/pipeline under scikit-learn syntax, specifically, any class with `fit` and `predict` methods. Furthermore, it includes interfaces to other relevant frameworks, notably PyTorch [64] and XGBoost [21]. By default, MachSMT is configured to use AdaBoost with 200 decision tree estimators and linear loss as its supervised learning core. Alternatively, it can be configured to use linear ridge regression or a multi-layer perceptron (MLP) deep neural network.

3.3 Configurations of MachSMT

MachSMT implements the following algorithm selection solutions.

1. **Mach-LogicEHM** – This configuration of MachSMT is analogous to the algorithm selection approach taken by SATZilla, with an EHM is constructed for every solver/logic pair. As state-of-the-art SMT solvers implement significantly different algorithms depending on the logic of the input problem, data points from different logics could negatively skew predictions.
2. **Mach-LogicPWC** – This configuration of MachSMT deploys the PairWise comparator approach as described in Section 2. In this configuration of

Feature	Description	Feature	Description
1–4	Frequency of constructs for problem description (`as`, `assert`, `check-sat`, `check-sat-assuming`)	52–87	Frequency of constructs of the theory of bit-vectors (e.g., `BitVec`, `bvor`, `bvadd`, `bvult`, ...)
5–13	Frequency of constructs for declaration and definition (e.g., `declare-const`, `define-fun`, ...)	88–133	Frequency of constructs of the theory of floating-point arithmetic (e.g., `FloatingPoint`, `RNE`, `fp.add`, `fp.leq`, `fp.isNaN`, ...)
14–15	Frequency of `echo` and `exit` constructs	134–148	Frequency of constructs of the theories of integers and reals (e.g., `Int`, `Real`, `>`, `+`, `*` `to_real`, `is_int`, ...)
16–27	Frequency of `get-*` constructs (e.g., `get-model`, `get-unsat-core`, ...)	149–184	Frequency of constructs of the theory of strings and regular expressions (e.g., `String`, `RegLan`, `str.len`, `str.++`, `re.diff`, ...)
28–29	Frequency of `push` and `pop` constructs	185	Avg. number of `select`s per array
30–31	Frequency of `reset` and `reset-assertions` constructs	186	Avg. `store` chain depth per array
32–34	Frequency of `set-*` constructs (`set-info`, `set-logic`, `set-option`)	187–189	Avg./Median/Deviation of the depth of BV adder chains
35–36	Frequency of `forall` and `exists` bindings	190–192	Number and ratio of `forall`/`exists` variables
37	Frequency of `let` bindings	193	Avg. quantifier nesting level
38–48	Frequency of core and Boolean constructs (e.g., `true`, `Bool`, `and`, `or`, `=>`, `ite`, ...)	194–195	Avg. arity and number of applications of uninterpreted functions
49–51	Frequency of constructs of the theory of arrays (`Array`, `select`, `store`)	196	Size of the smt2 file in bytes

Table 1: Complete list of the 196 features used in MachSMT.

the PWC, comparators are trained for every pair of solvers on the benchmarks of a common logic.

3. **Mach-Greedy** – In this configuration, the algorithm that performed the best over the training data is returned. Similar to Mach-LogicEHM and Mach-LogicPWC, it is filtered by the logic. This solution is a fallback for when MachSMT fails to learn a model that improves on the best standalone solver (as determined by the SMT-COMP results).

By default, MachSMT creates models for Mach-EHM and Mach-Greedy in the presence of one logic, and Mach-LogicEHM in case of multiple logics. In evaluation, MachSMT evaluates the performance of each approach on each logic. In deployment, MachSMT uses the approach that had the best-observed performance in evaluation.

3.4 Using MachSMT

MachSMT consists of three core tools, which are used to build, evaluate, and deploy MachSMT, respectively.

1. `machsmt_build` – This tool is the interface for building MachSMT's database around the solvers and benchmarks provided by the user. It takes as input a csv data file with columns *solver*, *benchmark*, and *score*. The solver column denotes the name of the solver, and the benchmark column indicates where the benchmark is located. We use `PAR-2` as score, with penalties for timeouts and incorrect answers (see Section 4.2). With default settings, the output of `machsmt_build` is a library directory containing the database and learnt models. For example, on input `data.csv` with output directory `dir`, `machsmt_build` is called as follows:

```
$ machsmt_build -f data.csv -l dir
```

2. `machsmt_eval` – This tool takes as input the library directory with the database and learnt models generated by `machsmt_build`, evaluates them under k-fold cross validation, and provides a summary of the results. `machsmt_eval` further tunes MachSMT to use the best empirically observed configuration based on the logic of the input benchmark. For example, on `dir`, it is called as follows:

```
$ machsmt_eval -l dir
```

3. `machsmt` – This tool is the primary interface to MachSMT's algorithm selection. Given an input benchmark and the library files generated above, `machsmt` will output a ranking of solvers, ordered from predicted to solve the input the fastest (top) to slowest (bottom). An example usage of `machsmt` on input file `example.smt2` with library directory `dir` is given in Figure 2.

3.5 Allocating Resources with MachSMT

In Figure 2, we present an example commandline usage of MachSMT primary prediction interface. We

```
$ machsmt example.smt2 -l dir
MachSMT selects: bitwuzla
    bitwuzla 81.102%
    cvc4 16.180%
    Z3 2.718%
```

Fig. 2 *Example Usage of MachSMT.*

suggest two different ways to allocate resources: either greedily spend all resources on the top selected solver or allocate resources based on the suggested allocation (see Section 5 for a comparison).

MachSMT utilizes a resource allocation scheme that leverages ideas from deep neural networks—specifically, *softmin*, an output activation function that computes the smooth minimum of, in our case, a vector $\vec{x}$ of predicted solver runtimes. Leveraging EHMs yields a predicted runtime x_i for every solver $s_i \in \mathcal{S}$. The regression models can be poor at predicting absolute ground truth runtime accuracy, but the predicted runtimes of the solver EHMs relative to each other, however, can be indicative for algorithm selection. We thus chose to compute the suggested resource allocation for a solver s_i as a softmin of x_i, defined as follows.

$$softmin(x_i) = \frac{e^{-x_i}}{\sum_{j=1}^{n} e^{-x_j}}$$

The softmin activation function normalizes predicted solver runtimes to predicted probabilities, and is inspired by classification in ML, where the *softmax* output activation function is frequently used to compute a smooth maximum. In contrast to softmax, softmin allows us to give preference to lower runtimes.

3.6 User-defined Features

MachSMT provides a simple interface for users to extend its set of predefined features with user-defined features. Each new feature is represented as a Python method that returns a single floating-point number (or an iterable that returns floating-point numbers). Features can be disabled, by having the user-defined procedure return `None`. All user-defined features are automatically included in building MachSMT and can significantly affect the accuracy of MachSMT when engineered to target a specific class of benchmarks.

4 Experimental Evaluation of MachSMT on SMT-COMP 2019 and 2020 Data

In this section, we present the evaluation of MachSMT, specifically on the benchmarks, solvers, and solver runtime analysis from SMT-COMP 2019 [37] and 2020 [9]. The results of this experiment are summarized in Tables 2 to 5 and Figure 3.

4.1 Experimental Setup and Methodology

In this experiment, we used the benchmarks, timing analysis, and solvers provided by the organizers of the SMT-COMP 2019 and 2020 competitions [9, 37]. The competition is organized in tracks, which are split into divisions. In 2019 and 2020, as in the years before [82], a division corresponded to an SMT-LIB logic. We consider all divisions of the Single Query (SQ) and Incremental (INC) tracks for each year. In the SQ track, each input only contains a single `check-sat` query, whereas in the INC track, inputs contain multiple such queries.

In both years, all solver input queries were performed on the StarExec computing service [76], which consists of a cluster of 2.4 GHz Intel Xeon machines running Red Hat Enterprise Linux 7.2. Each solver/benchmark pair was configured to have four cores and 60GB of memory available. The time limit for each pair was 2400 seconds in 2019, and 1200 seconds in 2020.

For this experiment, we configure MachSMT with AdaBoost with 200 decision tree estimators and linear loss as its supervised learning core. Further, we evaluate MachSMT in all of its configurations (as introduced in Section 3.3) using k-fold cross validation with $k = 5$. For cross-validation, the data set is randomly partitioned into k subsets per division and year. For more details on cross validation, see Section 2.

As we are leveraging data collected by the SMT-COMP organizers, we do not rerun with the allocation scheme as outlined in Section 3.5. Instead, we use whichever solver had the lowest predicted runtime. Note that this is usually slower than the allocation scheme.

4.2 Experimental Results

For each division, we compare MachSMT against the best solver of the division in the competition (including non-competing solvers). For the *Single Query* track, we evaluate solvers across, according to `PAR-2` scores [55]. Given the wallclock runtime w and the wallclock time limit t, we compute the `PAR-2` score as w on successful termination, $10 * t$ for an incorrect answer, and $2 * t$ for any other case of unsuccessful termination. For the

Logic in SQ'20	Best	PAR-2				
		Random	Best	VBS	Mach-LogicEHM	Impr. [%]
QF_BVFPLRA (153)	MathSAT5	48277.5	31111.1	159.5	**174.7**	**99.4**
LIA (299)	CVC4	393648.4	37071.9	2697.3	**12490.4**	**66.3**
BVFP (224)	CVC4	289843.2	266428.9	54434.9	**95958.1**	**64.0**
QF_UFBV (217)	Yices	35418.8	22343.0	5343.9	**9356.6**	**58.1**
UFDTNIRA (736)	Vampire	561429.7	177066.7	48076.2	**75373.3**	**57.4**
QF_LIA (2508)	CVC4	1246085.0	579953.4	169595.8	**269473.4**	**53.5**
BV (696)	CVC4	459663.8	239717.7	60146.9	**129250.2**	**46.1**
QF_UFNRA (27)	Yices	42957.1	12929.8	4670.0	**7902.6**	**38.9**
QF_ALIA (116)	Yices	39316.5	68.5	32.9	**45.8**	**33.2**
QF_NRA (2230)	Yices	1832304.9	1132073.9	468148.2	**812778.6**	**28.2**
AUFDTNIRA (300)	Vampire	268191.0	23710.6	5964.7	**17211.1**	**27.4**
QF_ABVFPLRA (41)	CVC4	15213.5	3269.6	30.0	**2436.3**	**25.5**
FP (993)	Z3	928925.1	462175.2	306571.7	**404901.1**	**12.4**
QF_IDL (834)	Yices	468813.2	234514.8	182994.3	**211312.8**	**9.9**
AUFDTLIRA (4878)	CVC4	4139948.8	1316952.2	1003100.6	**1189167.9**	**9.7**
UFDTLIRA (4296)	CVC4	4088170.6	1936992.8	1709161.2	**1769376.9**	**8.7**
QF_BV (6861)	Bitwuzla	1729172.7	431768.2	244154.8	**400073.9**	**7.3**
QF_ANIA (94)	MathSAT5	66033.4	617.9	251.1	**573.9**	**7.1**
UFNIA (5041)	Vampire	6813123.2	913033.2	603779.5	**861190.5**	**5.7**
AUFLIA (1310)	Vampire	1069749.1	507919.1	361485.5	**481708.6**	**5.2**
UFBV (72)	Z3	96086.8	55232.8	52830.0	**52832.7**	**4.3**
ABV (169)	Z3	357625.1	307762.2	295252.8	**297652.7**	**3.3**
QF_NIA (9195)	MathSAT5	10541986.4	7230607.6	4390297.4	**7024329.8**	**2.9**
AUFNIRA (300)	Vampire	627468.0	582179.9	549239.6	**566789.9**	**2.6**
QF_UFLIA (300)	Yices	74547.2	56.3	46.2	**54.9**	**2.5**
QF_RDL (247)	Yices	117274.8	83921.9	83164.3	**83598.5**	**0.4**
QF_UFIDL (300)	Yices	40814.8	5523.1	5490.4	**5508.2**	**0.3**
AUFLIRA (1346)	CVC4	796897.2	243525.3	228066.0	**242965.1**	**0.2**
FPLRA (27)	CVC4	55202.7	29740.3	29740.3	29740.3	0.0
BVFPLRA (185)	CVC4	338419.4	223227.4	223227.4	223227.4	0.0
NRA (93)	Z3	69642.8	7497.5	38.9	7498.0	0.0
ABVFP (30)	CVC4	40818.9	36003.5	36003.4	36003.5	0.0
UFFPDTLIRA (373)	CVC4	525984.4	103911.2	103911.2	103911.2	0.0
AUFBVDTLIA (683)	CVC4	1241067.5	791289.9	791289.9	791289.9	0.0
AUFFPDTLIRA (154)	CVC4	220808.6	48019.5	48019.5	48019.5	0.0
QF_UFLRA (432)	Yices	97099.5	4837.3	3287.3	4884.9	-1.0
QF_AX (300)	Yices	66061.7	6.3	5.3	6.5	-2.8
UFLIA (2278)	CVC4	3263274.9	2363827.7	2139934.2	2439073.8	-3.2
ABVFPLRA (75)	CVC4	158401.1	136806.3	136806.3	141602.0	-3.5
UFDT (1223)	CVC4	2296525.6	1990066.9	1889835.6	2059603.1	-3.5
QF_LRA (429)	OpenSMT	147136.2	71949.4	46349.1	76132.2	-5.8
UF (2291)	Vampire	4042624.6	3122146.8	2917476.7	3326149.0	-6.5
LRA (802)	Z3	788146.6	172873.6	71219.3	191840.8	-11.0
QF_BVFP (394)	Bitwuzla	20315.1	778.4	205.5	883.0	-13.4
QF_AUFBV (34)	Yices	42033.0	25818.9	20431.4	29298.1	-13.5
QF_UFNIA (300)	CVC4	152541.4	48058.9	12013.2	55528.3	-15.5
QF_FP (247)	Bitwuzla	134069.5	78980.8	42799.3	94053.4	-19.1
QF_ABV (3385)	Bitwuzla	126991.6	48643.1	42430.1	58977.6	-21.2
QF_SLIA (27033)	CVC4	4751221.5	698072.5	563735.7	889222.2	-27.4
AUFDTLIA (147)	CVC4	152649.5	23515.5	23255.2	31105.0	-32.3
QF_FPLRA (51)	COLIBRI	24654.4	7275.9	162.8	9663.8	-32.8
UFDTLIA (277)	CVC4	397736.7	250883.7	207326.4	355323.4	-41.6
QF_ABVFP (500)	Bitwuzla	66977.7	5460.8	3690.7	9589.7	-75.6
QF_UF (2800)	Yices	397239.9	388.8	325.8	724.5	-86.3
QF_S (927)	CVC4	69954.9	10415.2	4324.1	22949.6	-120.3
QF_AUFLIA (467)	Yices	228647.7	18.3	9.1	2412.6	-13058.7

Table 2: Results of Mach-LogicEHM on data from the SMT-COMP'20 SQ divisions.

Logic in INC'20	Best	PAR-2 Random	PAR-2 Best	PAR-2 VBS	PAR-2 Mach-LogicEHM	Impr. [%]
QF_ABV (376)	Yices	50028.7	12049.0	2283.9	**6493.4**	**46.1**
QF_AUFLIA (72)	Yices	24978.5	3293.0	2924.5	**2927.8**	**11.1**
QF_UFLIA (230)	Z3	158480.1	129340.4	107652.4	**117549.1**	**9.1**
UFLRA (748)	Z3	1131365.1	184002.0	159210.5	**174155.4**	**5.4**
UFNIA (2029)	Z3	3807401.9	2757190.6	1961676.0	2787434.9	-1.1
QF_LIA (69)	Yices	73247.5	45262.5	40039.5	45830.5	-1.3
QF_BVFP (51)	Bitwuzla	12529.7	371.3	89.1	379.5	-2.2
QF_UF (306)	Yices	2645.4	64.3	59.8	66.6	-3.5
UF (2031)	Z3	4034456.8	2391318.1	1692446.9	2612107.1	-9.2
AUFNIRA (165)	CVC4	146343.7	99766.7	60200.6	116514.8	-16.8
QF_AUFBV (30)	Yices	28754.0	14092.7	12687.6	17773.8	-26.1
QF_BV (484)	Bitwuzla	42345.7	20437.5	17019.1	25880.7	-26.6
QF_UFLRA (1223)	Z3	19339.8	8344.7	5198.8	10607.5	-27.1
QF_ALIA (44)	Z3	8597.9	702.3	398.0	1517.3	-116.1
QF_UFBV (932)	Bitwuzla	11434.8	1831.5	832.3	4574.3	-149.8

Table 3: Results of Mach-LogicEHM on data from the SMT-COMP'20 INC divisions.

Logic in SQ'19	Best	PAR-2 Random	PAR-2 Best	PAR-2 VBS	PAR-2 Mach-LogicEHM	Impr. [%]
QF_UFBV (223)	Yices	57238.9	26322.4	2536.0	**2891.0**	**89.0**
QF_UFNRA (26)	Yices	81791.6	20744.2	5635.6	**8119.3**	**60.9**
NRA (93)	Vampire	161246.9	53040.4	4891.9	**24018.8**	**54.7**
QF_NRA (2842)	Yices	4700323.3	3307748.9	918003.5	**1720350.2**	**48.0**
QF_FP (225)	COLIBRI	311524.5	252284.1	111486.3	**177289.9**	**29.7**
BV (823)	Q3B	775521.1	407336.7	129831.8	**294642.1**	**27.7**
QF_LIA (3136)	SPASS-SATT	3493282.0	493192.2	183339.1	**362941.9**	**26.4**
AUFLIRA (1683)	Z3	3302901.6	398577.1	316990.7	**327028.9**	**18.0**
QF_LRA (546)	SPASS-SATT	346010.5	128635.0	63374.3	**105550.3**	**17.9**
QF_AUFLIA (651)	Yices	528768.8	18.6	11.6	**15.3**	**17.5**
LIA (299)	Z3	749636.7	14.8	11.3	**12.9**	**13.1**
AUFLIA (1638)	Vampire	3423183.4	1242230.7	851466.2	**1104767.7**	**11.1**
UFNIA (6253)	CVC4	19759215.9	13320056.8	7804961.4	**11927375.3**	**10.5**
QF_IDL (1042)	Z3	1477558.6	557543.6	437743.1	**509547.6**	**8.6**
AUFNIRA (300)	CVC4	1302356.6	1171565.6	1110988.5	**1155968.5**	**1.3**
FP (1238)	Z3	2470540.0	1006063.1	719531.7	**1000203.6**	**0.6**
QF_UFNIA (300)	CVC4	260047.3	72279.0	19207.9	**72091.2**	**0.3**
QF_RDL (247)	Yices	182890.5	162609.5	162070.6	**162327.5**	**0.2**
UFDT (1547)	CVC4	5342330.5	5011100.6	4601262.4	**5004303.6**	**0.1**
QF_UFIDL (300)	Yices	62584.7	10206.4	10172.4	**10197.5**	**0.1**
QF_NIA (11494)	CVC4	28208367.0	17006943.3	10126265.9	17009089.1	0.0
UFBV (72)	Z3	179316.1	110434.2	100836.9	110438.1	0.0
QF_AX (300)	Yices	107798.6	4.8	4.6	4.8	-0.3
QF_AUFBV (41)	Yices	39919.8	14811.6	6250.6	14900.5	-0.6
QF_BVFP (516)	CVC4	25474.1	8997.5	3643.2	9063.2	-0.7
QF_UFLIA (300)	Yices	193160.2	31.6	19.9	32.0	-1.4
UF (2816)	Vampire	10822123.3	8052636.1	7371252.1	8180674.7	-1.6
UFLIA (2848)	CVC4	9383261.2	5857081.7	5426481.9	5967471.7	-1.9
QF_ABV (7538)	Poolector	236916.1	104633.3	84039.8	107399.8	-2.6
AUFDTLIA (275)	CVC4	191254.3	46638.2	23377.1	51228.3	-9.8
QF_BV (8909)	Poolector	4757776.7	868489.1	549558.4	1021521.2	-17.6
LRA (1003)	Z3	2022790.0	363124.4	223245.7	473842.6	-30.5
UFDTLIA (299)	CVC4	752499.4	495667.2	434393.8	725651.2	-46.4
QF_UFLRA (541)	SMTInterpol	280508.4	3214.4	1964.0	4972.3	-54.7
QF_UF (3512)	Yices	1098010.6	422.4	272.8	1451.2	-243.6
QF_ALIA (139)	Yices	156602.7	57.9	50.1	4852.9	-8274.6

Table 4: Results of Mach-LogicEHM on data from the SMT-COMP'19 SQ divisions.

Logic in INC'19	Best	PAR-2 Random	PAR-2 Best	PAR-2 VBS	Mach-LogicEHM	Impr. [%]
QF_UFLIA (302)	Z3	377725.6	343663.9	248671.6	**267783.4**	**22.1**
QF_ABV (469)	Yices	165761.3	13054.3	3727.9	**11461.8**	**12.2**
QF_UFLRA (1529)	Z3	33688.3	11937.7	7505.0	**10653.4**	**10.8**
QF_AUFLIA (72)	Yices	7915.0	3029.5	2901.4	**2936.0**	**3.1**
UFLRA (935)	Z3	3355019.8	223811.2	214824.6	**223677.4**	**0.1**
QF_UF (383)	Yices	341.2	78.2	78.0	78.2	0.0
QF_LIA (69)	Yices	120316.9	72456.8	68228.8	73274.1	-1.1
AUFNIRA (165)	CVC4	465185.6	183826.9	109727.3	200152.9	-8.9
QF_AUFBVNIA (32)	MathSAT	53475.0	849.6	667.0	1003.1	-18.1
QF_BV (583)	Boolector	270820.1	40616.1	29793.1	59064.0	-45.4
QF_UFBV (1165)	Boolector	14560.1	3261.0	638.1	7311.1	-124.2
QF_ALIA (44)	Z3	3440.9	854.5	394.1	5334.1	-524.3

Table 5: Results of Mach-LogicEHM on data from the SMT-COMP'19 INC divisions.

Incremental track, we use $w + (2 * \frac{t}{n}) * (n - m)$, with n the total number of `check-sat` calls in the input and m the total number of `check-sat` calls successfully solved.

The results of this experiment are given in Tables 2 to 5. We consider all divisions that contain at least 25 benchmarks and exclude incremental benchmarks with only a single `check-sat` query. We further consider three baselines when evaluating MachSMT: random algorithm selection (Random), the best solver of the division (Best), and the virtual best solver (VBS). Tables 2 to 5 give the `PAR-2` score for each configuration, as well as the percent improvement of Mach-LogicEHM over the best solver of each division. We only report the results for Mach-LogicEHM as it is, overall, the best performing configuration. Note that the VBS corresponds to perfect algorithm selection and can therefore not be beaten. We observe that overall and across all four tracks, MachSMT improves on the best solver in 57 (out of 119) divisions— for SQ'20 on 28 (out of 56), for INC'20 on 4 (out of 15), for SQ'19 on 20 (out of 36), and for INC'19 on 5 (out of 12).

We present selected cumulative distribution function (CDF) plots of the top three results with the most improvement in the SQ'19 and SQ'20 tracks in Figure 3. In this context, a CDF visualizes how a solver performs on a database of inputs. A point (x,y) denotes that a solver $\mathcal{S}$ solves y inputs within x seconds each.

4.3 Discussion

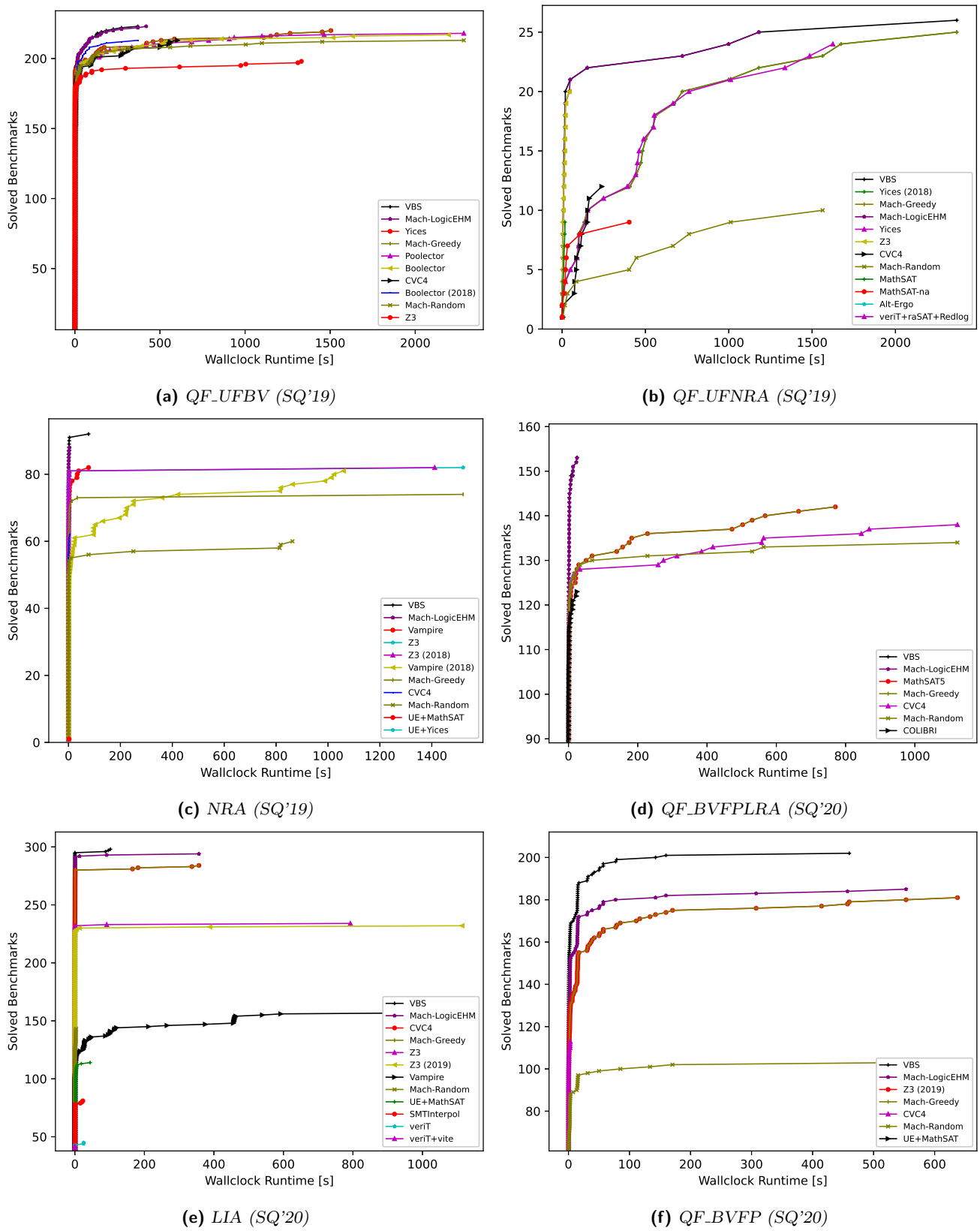
In Section 3.3, we describe three configurations of MachSMT. In our evaluation of SMT-COMP data, we observe that Mach-LogicEHM performs significantly better than the other MachSMT configurations. Our experimental results confirm that algorithm selection (in particular through the use of EHMs) can be a powerful way to address the combinatorial explosion that solver users face when trying to decide which solver-configuration pair is best suited for their application.

We note that MachSMT is particularly powerful in the context of divisions with a diverse set of solving procedures, e.g., QF_BVFPLRA and QF_UFBV. Possible reasons for divisions where MachSMT performs worse than the best performing solver are too few (or too homogeneous benchmarks), lack of diversity of the algorithms to select, and the feature vector not sufficiently capturing the challenges of an input problem. For example, an interesting observation occurs in QF_BV logics of the SQ tracks, on a specific set of small benchmarks, a benchmark family originating from [61]. This benchmark family encodes bit-vector rewrites, which are trivial if a solver implements these specific rewrites. However, MachSMT is tricked into picking the competition winner Bitwuzla, even though it misses some of these rewrites and performs worse on some of these benchmarks than other solvers. This is mainly due to the fact that these benchmarks are very small in size, and Bitwuzla usually outperforms other solvers on instances of small input size.

As noted in previous work, algorithm selection methods work well for non-homogeneous benchmarks, especially where no single algorithm performs best across the board. EHMs are an effective way to distinguish between such algorithms and predict which one might perform the best on a given instance.

The best algorithm selection methods were based on EHMs. We found that Mach-LogicEHM outperformed Mach-LogicPWC by a median `PAR-2` score improvement of 59.8%. Over Mach-Greedy, on the other hand, Mach-LogicEHM has a median `PAR-2` score improvement of 11.8%.

One major threat to the validity of any ML solution is the generalizability of the learnt models on unseen data. Previous work has noted that a practical



(a) *QF_UFBV (SQ'19)*

(b) *QF_UFNRA (SQ'19)*

(c) *NRA (SQ'19)*

(d) *QF_BVFPLRA (SQ'20)*

(e) *LIA (SQ'20)*

(f) *QF_BVFP (SQ'20)*

Fig. 3 *Comparison of MachSMT, the virtual best solver, and participating solvers in divisions of the SMT-COMP 2019 and SMT-COMP 2020 Single Query (SQ) Tracks.*

way to address this issue is to use $k-$fold cross validation scheme [56, 71], thus motivating our use of this approach in our experiments. We further note that our evaluation of MachSMT on SMT-COMP data includes decades of runtime analysis and more than 100 GB of benchmarks spanning numerous applications, giving us greater confidence in the robustness of our results.

5 Experimental Results for Solver Configuration Selection

In this section, we use MachSMT to predict a ranking and resource allocation for a set of configurations of a single solver to be used in a sequential portfolio setting.

5.1 Experimental Setup and Methodology

In this experiment, we evaluate MachSMT for logics with uninterpreted functions (UF) and quantifiers on 23 different configurations of the SMT solver cvc5 [8], as taken from its SMT-COMP'21 script[2]. We use all benchmarks available in these logics from the 2021 release of SMT-LIB [10], the number of benchmarks for each logic is given in parenthesis in Table 6.

This experiment differs significantly from the experiment performed in Section 4 in the sense that all 23 configurations are configurations of a single solver—they only differ in their configured options, which enable or disable different algorithms and heuristics. In the competition setting, each of these configurations is ran sequentially, each with a specific time limit, until a solution is found or the overall time limit is exceeded. Resource allocations (the time limit) per configuration have been determined manually, and are configured based on the overall time limit of the competition. This sequential portfolio setting is encoded in the aforementioned competition script of cvc5.

Our goal for this experiment is to utilize MachSMT to predict a ranking and resource allocations for these configurations in a sequential setting that outperforms cvc5's competition script setting.

In Section 4, we evaluated MachSMT and all of its configurations (as introduced in Section 3.3) and observed that Mach-LogicEHM performs best. Hence, for this experiment, we exclusively consider this configuration. As discussed in Section 3, we suggest two resource allocation strategies based on the predictions produced by MachSMT. First, allocating all resources to the solver with the highest predicted performance. This was the only considered allocation strategy used in Section 4. Second, we use the resource allocation outlined in Section 3.5, where we compute a softmin over the predicted runtimes produced by the EHMs. We refer to these two configurations of MachSMT as *MachSMT* and *MachSMT–alloc*, respectively.

Our experimental setup is analogous to Section 4. However, for collecting the training data we ran all 23 solver configurations with a time limit of 60 seconds. We performed all experiments in this section on a cluster running Ubuntu 16.04 with Intel Xeon E5-2620 CPUs with 2.1GHz and 128GB memory. At runtime, for each solver/benchmark pair we used a time limit of 1200 seconds and a memory limit of 8GB. Further, we configure MachSMT's supervised learning core to train on an 8 layer multi-layer perceptron (MLP) neural network of 128 neurons each. We use ReLU activation functions with batch normalization applied after each layer trained on an NVIDIA 2070 GPU for 30 minutes.

5.2 Experimental Results

The overall results of this experiment are presented in Table 6, which gives the `PAR-2` score and the number of solved instances for each configuration and considered logic. Figure 4 shows CDF plots of selected logics. In total, we observe that MachSMT-alloc solves 898 more benchmarks than the competition version of cvc5, and improves by 5% in total `PAR-2` score. Further, MachSMT is able to improve over the carefully hand-tuned sequential portfolio of the cvc5 competition script on the majority of the quantified logics—achieving a lower `PAR-2` score in 13 out of 16 logics while solving more benchmarks in 14 out of 16 logics with up to a 93.4% `PAR-2` improvement in the AUFDTLIA logic.

6 Experimental Results on Network Protocols Problems with Domain Specific Features

In this section, we present the evaluation of MachSMT on a network application, and show how domain specific knowledge can be utilized to extend MachSMT to further improve performance. The results of this experiment are summarized in Figure 5.

6.1 Experimental Setup and Methodology

In this section, we evaluate MachSMT's abilities to leverage basic domain specific knowledge to increase the

[2] https://github.com/cvc5/cvc5/blob/master/contrib/competitions/smt-comp/run-script-smtcomp2021

Logic	PAR-2			Solved		
	cvc5	MachSMT	MachSMT–alloc	cvc5	MachSMT	MachSMT–alloc
ALIA (42)	**2406.7**	2691.3	2921.7	41	41	41
AUFBV (1522)	2689925.2	**2608537.5**	2608683.6	410	**448**	446
AUFBVDTLIA (1690)	2403641.5	**1961063.1**	1967667.0	716	**876**	873
AUFBVFP (57)	90215.3	86963.7	**85188.4**	20	21	**22**
AUFDTLIA (728)	79737.0	**5190.7**	5264.2	698	**728**	**728**
AUFLIA (3276)	1422835.2	**1353325.9**	1353915.0	2695	2724	**2726**
AUFLIRA (20011)	**626978.9**	755386.7	760748.0	19755	**19766**	**19766**
AUFNIRA (1480)	1016878.9	1016541.9	**1008664.9**	1058	1060	**1064**
UF (7590)	9420855.1	8576358.4	**8573813.0**	3744	4047	**4053**
UFBVLIA (208)	497350.9	**494424.3**	494424.5	1	**2**	**2**
UFDT (4569)	5333216.9	5090884.3	**5074775.5**	2380	2463	**2471**
UFDTLIA (327)	381207.6	215555.2	**204909.8**	183	238	**243**
UFFPDTLIRA (774)	192772.6	**182239.3**	187409.0	696	**701**	699
UFIDL (68)	24515.3	**24402.7**	24408.9	58	58	58
UFLIA (10127)	**6041256.6**	6115048.8	6077710.4	7625	7610	**7630**
UFNIA (13463)	13258581.0	13089905.7	**12933659.8**	7988	8067	**8144**
Total (65932)	43482374.7	41578519.5	**41364163.7**	48068	48850	**48966**

Table 6: Comparison of MachSMT, MachSMT–alloc and the cvc5 competition script on all evaluated logics.

overall prediction accuracy of learnt models and improve MachSMT's overall performance. For this experiment, we consider network verification problems, specifically, firewall and router queries based on the SecGuru tool by Microsoft Research [43].

We extend MachSMT via its user-defined feature interface (outlined in Section 3.6) to capture the basic attributes of the underlying query. Specifically, we include the following domain-specific features: categorical features denoting the benchmark type (firewall or router), number of rules, number of allow rules, number of deny rules, block rules, IP range width, and port range width. We train and test on two separate data sets of 5,000 SecGuru benchmarks.

We consider three underlying solvers: Bitwuzla [59], cvc5 [8], and Z3 [57]. All benchmarks produced are of the `QF_BV` logic. We baseline against MachSMT without the domain-specific features. Both variants of MachSMT are using Mach-LogicEHM mode for algorithm selection.

All experiments in this section were performed on the Compute Canada computing service [6], a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz with 8 GB of memory with a wallclock runtime of 10 minutes. Wallclock runtimes are rounded to the nearest second.

This data set was collected offline, and we do not rerun with the allocation scheme as outlined in Section 3.5. Instead, we use whichever solver had the lowest predicted runtime. Note, however, that this is usually slower than the allocation scheme. In this experiment, we configure MachSMT's supervised learning core to use linear ridge regression with cross-validation on the regularization parameter[3].

6.2 Experimental Results

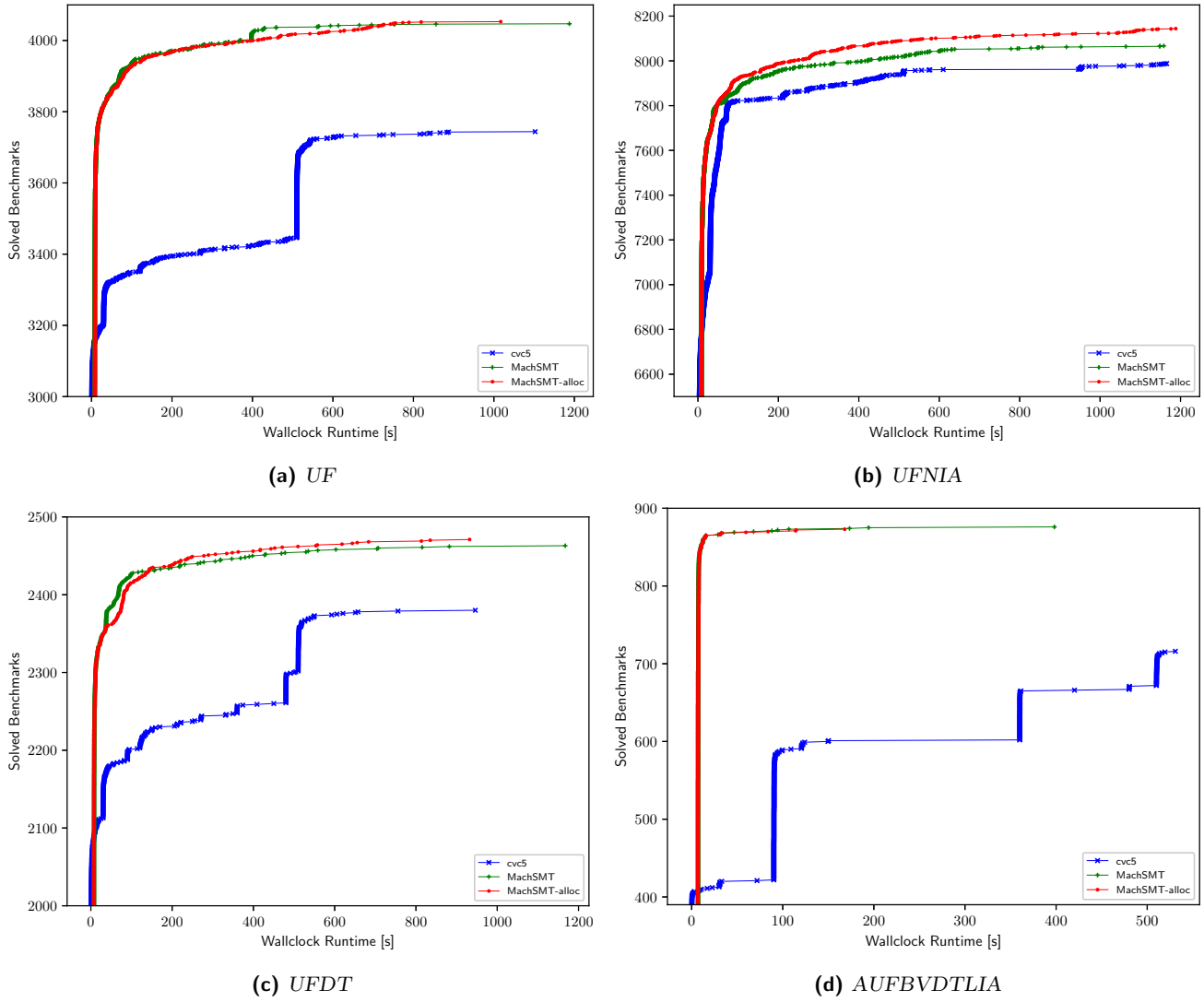
The results of this experiment are presented as a CDF plot in Figure 5. Overall, we observe MachSMT to improve on the best standalone solver Bitwuzla by 42.5% without the domain-specific features, and by 77.3% with domain-specific features in `PAR-2` score. The only difference between the two configurations of MachSMT is the inclusion of the additional features—the domain-specific features account for 34.8% of the total improvement despite their simplicity.

7 Analysis of Experimental Results

One major threat to the validity of any ML solution is the generalizability of the learnt models on unseen data. In the context of MachSMT, this scenario arises when asking MachSMT to solve classes of benchmarks that are divergent from the ones it has been trained on. In particular, in Section 4 we deployed $k-$fold cross-validation, which partitions the training and testing set. This scheme and its impacts on accuracy, bias, and error have been studied extensively [56, 71].

Furthermore, logics in SMT-LIB are organized in families, depending on the source and application of the benchmarks. Some families contain a small number of benchmarks, and some logics contain a small

[3] sklearn.linear_model.RidgeCV



(a) *UF* **(b)** *UFNIA*

(c) *UFDT* **(d)** *AUFBVDTLIA*

Fig. 4 *Comparison of the two different MachSMT resource allocation schemes against cvc5 competition script.*

number of families. Both scenarios may result in MachSMT underperforming on a new (or underrepresented) benchmark set when using the models built on SMT-COMP data. Generally, when similar data is not available or underrepresented in the SMT-COMP models, users are strongly encouraged to include runtime data for their new benchmarks and rebuild MachSMT. Note that algorithm selection in the context of floating-point problems has been shown to generalize on industrial benchmarks from an independently fuzzed, synthetic data set [75]. This may be an alternative approach in cases where not enough benchmarks are available. Alternatively, it may be beneficial to merge related logics (similarly to what has been done in SMT-COMP to define divisions since 2021) to achieve larger data sets.

Determining feature importance to improve the efficiency of ML is an interesting and challenging question, MachSMT included. We are aware of two existing strategies: blackbox analysis on the input-output behaviour (e.g., SHAP [53]). Another way is to leverage model-specific attributes if possible. We do not consider the former in this paper, and the latter is only applicable to certain types of models.

In this paper, we considered 3 types of ML models: AdaBoosting (Section 4), multi-layer perceptron (MLP) neural network (Section 5), and Linear Ridge regression (Section 6). However, only the latter has a way to extract feature importance directly (via the weights). In this experiment, we noticed that the weights of the trained model were significantly higher for the domain-specific features, in particular the number of allowable rules and bit-vector multiplication.

By far, the best performing configuration of MachSMT is Mach-LogicEHM. Notably, Mach-LogicEHM out-

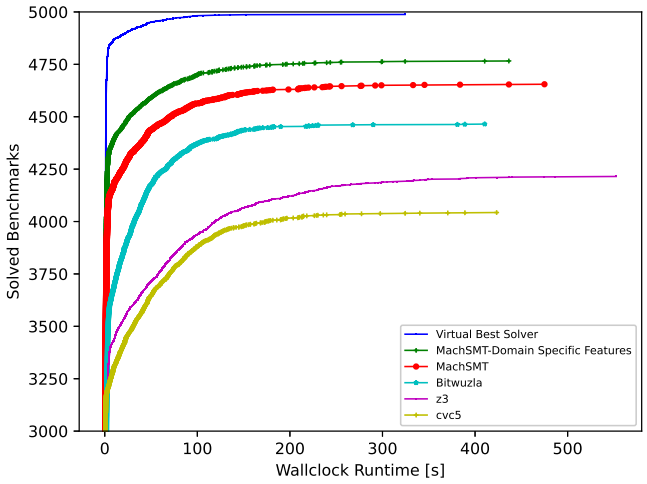


Fig. 5 *MachSMT with and without domain-specific knowledge on Network Verification Problems from SecGuru.*

performed Mach-LogicPWC and Mach-Greedy by a median `PAR-2` score improvement of 59.8% and 11.8%. From a regression standpoint, the overall model accuracy of precisely predicting the runtime for a benchmark is quite poor. However, even though individual predictions might not reflect the ground truth runtime, the relative ranking of the predicted runtimes can be highly accurate, as observed in the paper.

The artifact for this paper is available at https://zenodo.org/record/7383299.

8 Related Work

In this section, we provide an overview of previous work on algorithm selection in the context of constraint solvers and contrast it with MachSMT.

8.1 Key differences between SATZilla and MachSMT

As mentioned above, SATZilla was the first algorithm selection method in the context of logic solvers [86]. While SATZilla inspires our work, MachSMT differs from SATZilla in several key ways. First, SATZilla deploys a feature selection scheme to avoid the curse of dimensionality. While good in practice in the SAT setting, feature selection does lose a significant amount of information. Further, it can be costly to compute optimal feature subsets.

In contrast, MachSMT leverages a learnt dimensionality reduction scheme, namely, Principal Component Analysis (PCA). The key advantage of PCA is that it does not perform a search for an optimal feature subset (as is required in the context of feature selection) and is thus significantly more efficient. Further, MachSMT deploys a modern ML pipeline, including an ensemble learning approach, namely Adaptive Boosting [26].

8.2 Algorithm Selection for Logic Solvers and Their Applications

Algorithm selection tools have a rich history and have been around at least since 1976, when Rice was the first to propose the technique [68]. Algorithm selectors have been extensively used in many contexts, e.g., in classifiers for machine learning [3], combinatorics [46], and other NP-hard optimization problems [62, 78, 80]. Within the context of constraint solvers, algorithm selectors have been proposed for QBF [54, 67], SAT [86, 87, 88], CSP solvers [4, 30, 42, 47], and recommenders for ATP tools [77, 79].

In the setting of SMT solver applications, symbolic execution tools have used algorithm selection strategies [83] and portfolio strategies [40] for the specific classes of instances within the context of the bit-vector theory. This would be an ideal use case of MachSMT since we provide a complete solution.

There have been other works using machine learning to improve the performance of SMT solvers. Balunovic et al. [7] use neural networks and synthesis to find tactics and strategies for three SMT-LIB theories. A previous version of our work proposed an algorithm selection tool for the QF_FP theory [75]. To the best of our knowledge, MachSMT is the first publicly available tool for the entirety of SMT-LIB. Other works have leveraged machine learning to improve internal heuristics in solvers [15, 52, 69].

Pairwise ranking has been used in algorithm selection in the latest versions of SATZilla [89] and in variable selection in the context of splitting heuristics in divide-and-conquer parallel SAT solvers [58].

Recently, Pimpalkhare et al. released a system for dynamic algorithm selection on SMT solvers [66]. Their paper directly compares against MachSMT over four logics considered in Section 4. The advantage of a system like this is when runtime data is unavailable, and data must be collected online. However, their reinforcement learning approach still requires significant exploration to be predictive, and, in principle, MachSMT could also leverage the runtime data they collect. Nevertheless, in their evaluation, MachSMT outperformed their system when runtime data was provided.

9 Conclusions and Future Work

In this paper, we presented MachSMT, the first algorithm selection tool that spans the entirety of the SMT-LIB logics. MachSMT is designed to be user-friendly and easily modifiable by users for their specific application and SMT solvers of interest.

We extensively evaluated MachSMT over several usage scenarios and empirically demonstrated its efficiency and efficacy. Using MachSMT, we observe improvement in 57 out of 119 divisions in all tracks from the SMT-COMP '19 and '20, with up to a 99.4% improvement in `PAR-2` score over the best performing solver for the QF_BVFPLRA SQ'20 division.

We further evaluated MachSMT to predict a ranking and resource allocation for 23 configurations used in the cvc5 competition script and observed that MachSMT was able to solve 898 more benchmarks with up to an 93.4% improvement in `PAR-2` score. Finally, we evaluated MachSMT on network verification problems with simple domain-specific knowledge and observed an improvement of 77.3% in `PAR-2` score.

For future work, we plan to extend our feature set with more (theory-)specific features based on feedback from the SMT community. Recent research on online data collection strategies has increased the usability and pragmatism of algorithm selection [66]. Furthermore, the feature vector in MachSMT has significant room for improvement. One up-and-coming candidate for a highly representative feature vector is using learnt features (i.e., graph neural networks) over the graphical syntactical structure of the logic of the problem at hand. This was recently studied by Hula et al. [41] with promising results.

10 Acknowledgements

We would like to thank Karthick Jayaraman for his assistance in obtaining the benchmarks used in Section 6. We would like to thank Vineel Nagisetty, Laura Graves, Murphy Berzish, Federico Mora, and the SMT and TACAS communities for their feedback on earlier versions of this work.

References

1. MachSMT GitHub repository. https://github.com/machsmt/machsmt (2022)
2. Agrawal, S., Agrawal, J.: Survey on anomaly detection using data mining techniques. Procedia Computer Science **60**, 708–713 (2015). DOI 10.1016/j.procs.2015.08.220
3. Ali, S., Smith, K.A.: On learning algorithm selection for classification. Appl. Soft Comput. **6**(2), 119–138 (2006). DOI 10.1016/j.asoc.2004.12.002
4. Amadini, R., Gabbrielli, M., Mauro, J.: SUNNY: a lazy portfolio approach for constraint solving. Theory Pract. Log. Program. **14**(4-5), 509–524 (2014). DOI 10.1017/S1471068414000179
5. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: N. Bjørner, A. Gurfinkel (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, pp. 1–9. IEEE (2018). DOI 10.23919/FMCAD.2018.8602994
6. Baldwin, S.: Compute canada: advancing computational research. In: Journal of Physics: Conference Series, vol. 341, p. 012001. IOP Publishing (2012)
7. Balunovic, M., Bielik, P., Vechev, M.T.: Learning to solve SMT formulas. In: S. Bengio, H.M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pp. 10338–10349 (2018). URL http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas
8. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: D. Fisman, G. Rosu (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 13243, pp. 415–442. Springer (2022). DOI 10.1007/978-3-030-99524-9_24. URL https://doi.org/10.1007/978-3-030-99524-9_24
9. Barbosa, H., Hyvärinen, A., Hoenecke, J.: Smt-comp 2020. https://www.smt-comp.org/2020 (2020)
10. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2020)

11. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: A. Gupta, D. Kroening (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
12. Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England), vol. 13, p. 14 (2010)
13. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: G. Gopalakrishnan, S. Qadeer (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6806, pp. 171–177. Springer (2011). DOI 10.1007/978-3-642-22110-1_14
14. Ben Khadra, M.A., Stoffel, D., Kunz, W.: gosat: Floating-point satisfiability as global optimization. In: D. Stewart, G. Weissenbacher (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017, pp. 11–14. IEEE (2017). DOI 10.23919/FMCAD.2017.8102235
15. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features - A simple but effective approach **11245**, 144–159 (2018). DOI 10.1007/978-3-030-03421-4_11
16. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J.R., Maillard, K., Pan, J., Parno, B., Protzenko, J., Ramananandro, T., Rane, A., Rastogi, A., Swamy, N., Thompson, L., Wang, P., Béguelin, S.Z., Zinzindohoue, J.K.: Everest: Towards a verified, drop-in replacement of HTTPS. In: B.S. Lerner, R. Bodík, S. Krishnamurthi (eds.) 2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA, *LIPIcs*, vol. 71, pp. 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). DOI 10.4230/LIPIcs.SNAPL.2017.1. URL https://doi.org/10.4230/LIPIcs.SNAPL.2017.1
17. Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods Syst. Des. **45**(2), 213–245 (2014). DOI 10.1007/s10703-013-0203-7
18. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: T. Vojnar, L. Zhang (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 11427, pp. 79–98. Springer (2019). DOI 10.1007/978-3-030-17462-0_5
19. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: T. Vojnar, L. Zhang (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 11427, pp. 79–98. Springer (2019). DOI 10.1007/978-3-030-17462-0_5
20. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. **12**(2), 10:1–10:38 (2008). DOI 10.1145/1455518.1455522
21. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, pp. 785–794. ACM, New York, NY, USA (2016). DOI 10.1145/2939672.2939785. URL http://doi.acm.org/10.1145/2939672.2939785
22. Christ, J., Hoenicke, J., Nutz, A.: Smtinterpol: An interpolating SMT solver. In: A.F. Donaldson, D. Parker (eds.) Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7385, pp. 248–254. Springer (2012). DOI 10.1007/978-3-642-31759-0_19. URL https://doi.org/10.1007/978-3-642-31759-0_19
23. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: N. Piterman, S.A. Smolka (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7795, pp. 93–107. Springer (2013). DOI 10.1007/978-3-642-36742-7_7
24. Drucker, H.: Improving regressors using boosting techniques. In: D.H. Fisher (ed.) Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997, pp. 107–115. Morgan Kaufmann (1997)
25. Dutertre, B.: Yices 2.2. In: A. Biere, R. Bloem (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8559, pp. 737–744. Springer (2014). DOI 10.1007/978-3-319-08867-9_49
26. Freund, Y., Schapire, R., Abe, N.: A short introduction to boosting. Journal-Japanese Society For Artificial Intelligence **14**(771-780), 1612 (1999)
27. Fu, Z., Su, Z.: Xsat: A fast floating-point satisfiability solver. In: S. Chaudhuri, A. Farzan (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 9780, pp. 187–209. Springer (2016). DOI 10.1007/978-3-319-41540-6_11
28. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution). In: D. Beyer, M. Huisman, F. Kordon, B. Steffen (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III, *Lecture Notes in Computer Science*, vol. 11429, pp. 209–213. Springer (2019). DOI 10.1007/978-3-030-17502-3_15
29. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: W. Damm, H. Hermanns (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4590, pp. 519–531. Springer (2007). DOI 10.1007/978-3-540-73368-3_52
30. Gent, I.P., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Learning when to use lazy learning in constraint solving. In: H. Coelho, R. Studer, M.J. Wooldridge (eds.) ECAI 2010 - 19th

European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings, *Frontiers in Artificial Intelligence and Applications*, vol. 215, pp. 873–878. IOS Press (2010). DOI 10.3233/978-1-60750-606-5-873
31. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). DOI 10.1145/2093548.2093564
32. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
33. Goues, C.L., Leino, K.R.M., Moskal, M.: The boogie verification debugger (tool paper). In: G. Barthe, A. Pardo, G. Schneider (eds.) Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 7041, pp. 407–414. Springer (2011). DOI 10.1007/978-3-642-24690-6_28
34. Greenland, S., Mansournia, M.A., Altman, D.G.: Sparse data bias: a problem hiding in plain sight. bmj **352**, i1981 (2016). DOI 10.1136/bmj.i1981
35. Grira, N., Crucianu, M., Boujemaa, N.: Unsupervised and semi-supervised clustering: a brief survey. A review of machine learning techniques for processing multimedia content **1**, 9–16 (2004)
36. Guidotti, D., Barrett, C., Katz, G., Pulina, L., Narodyska, N., Tacchella, A.: The VNN-LIB standard. URL http://www.vnnlib.org/wp-content/uploads/2020/07/main-1.pdf
37. Hadarean, L., Hyvärinen, A., Niemetz, A., Reger, G.: Smt-comp 2019. https://www.smt-comp.org/2019 (2019)
38. Halko, N., Martinsson, P., Tropp, J.A.: Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev. **53**(2), 217–288 (2011). DOI 10.1137/090771806
39. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., et al.: Array programming with numpy. Nature **585**(7825), 357–362 (2020)
40. Healy, A., Monahan, R., Power, J.F.: Predicting SMT solver performance for software verification. In: C. Dubois, P. Masci, D. Méry (eds.) Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016, *EPTCS*, vol. 240, pp. 20–37 (2016). DOI 10.4204/EPTCS.240.2
41. Hula, J., Mojzísek, D., Janota, M.: Graph neural networks for scheduling of SMT solvers. In: 33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021, Washington, DC, USA, November 1-3, 2021, pp. 447–451. IEEE (2021). DOI 10.1109/ICTAI52525.2021.00072. URL https://doi.org/10.1109/ICTAI52525.2021.00072
42. Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: A hierarchical portfolio of solvers and transformations. In: H. Simonis (ed.) Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8451, pp. 301–317. Springer (2014). DOI 10.1007/978-3-319-07046-9_22
43. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Microsoft Research pp. 1–11 (2014)
44. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: R. Majumdar, V. Kuncak (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 10426, pp. 97–117. Springer (2017). DOI 10.1007/978-3-319-63387-9_5
45. Kira, K., Rendell, L.A.: A practical approach to feature selection. In: D.H. Sleeman, P. Edwards (eds.) Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992, pp. 249–256. Morgan Kaufmann (1992). DOI 10.1016/b978-1-55860-247-2.50037-1
46. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: C. Bessiere, L.D. Raedt, L. Kotthoff, S. Nijssen, B. O'Sullivan, D. Pedreschi (eds.) Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach, *Lecture Notes in Computer Science*, vol. 10101, pp. 149–190. Springer (2016). DOI 10.1007/978-3-319-50137-6_7
47. Kotthoff, L., Gent, I.P., Miguel, I.: An Evaluation of Machine Learning in Algorithm Selection for Search Problems. AI Commun. **25**(3), 257–270 (2012). DOI 10.3233/AIC-2012-0533. URL https://doi.org/10.3233/AIC-2012-0533
48. Kwon, D., Kim, H., Kim, J., Suh, S.C., Kim, I., Kim, K.J.: A survey of deep learning-based network anomaly detection. Clust. Comput. **22**(Suppl 1), 949–961 (2019). DOI 10.1007/s10586-017-1117-8
49. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. nature **521**(7553), 436–444 (2015)
50. Leino, K.R.M.: Automating theorem proving with SMT. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7998, pp. 2–16. Springer (2013). DOI 10.1007/978-3-642-39634-2_2
51. Li, X., Wang, L., Sung, E.: A study of adaboost with svm based weak learners. In: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 1, pp. 196–201. IEEE (2005)
52. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: N. Creignou, D.L. Berre (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9710, pp. 123–140. Springer (2016). DOI 10.1007/978-3-319-40970-2_9
53. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. In: I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (eds.) Advances in Neural Information Processing Systems 30, pp. 4765–4774. Curran Associates, Inc. (2017). URL http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf
54. Malitsky, Y.: Evolving instance-specific algorithm configuration. In: Instance-Specific Algorithm Configuration, pp. 93–105. Springer (2014). DOI 10.1007/978-3-319-11230-5. URL https://doi.org/10.1007/978-3-319-11230-5
55. Marijn Heule Matti Järvisalo, M.S.: Sat race 2019 (2019). URL http://sat-race-2019.ciirc.cvut.cz/
56. Moore, A.W.: Cross-validation for detecting and preventing overfitting. School of Computer Science Carneigie Mellon University (2001)

57. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008). DOI 10.1007/978-3-540-78800-3_24
58. Nejati, S., Frioux, L.L., Ganesh, V.: A machine learning based splitting heuristic for divide-and-conquer solvers. In: H. Simonis (ed.) Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings, *Lecture Notes in Computer Science*, vol. 12333, pp. 899–916. Springer (2020). DOI 10.1007/978-3-030-58475-7_52
59. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020). URL https://arxiv.org/abs/2006.01621
60. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). DOI 10.3233/sat190101
61. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C.W., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. In: M. Janota, I. Lynce (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings, *Lecture Notes in Computer Science*, vol. 11628, pp. 279–297. Springer (2019). DOI 10.1007/978-3-030-24258-9_20. URL https://doi.org/10.1007/978-3-030-24258-9_20
62. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. In: Irish conference on artificial intelligence and cognitive science, pp. 210–216 (2008). URL https://homepages.laas.fr/ehebrard/papers/aics2008.pdf
63. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transf. **11**(4), 339–353 (2009). DOI 10.1007/s10009-009-0118-1
64. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32**, 8026–8037 (2019)
65. 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., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
66. Pimpalkhare, N., Mora, F., Polgreen, E., Seshia, S.A.: Medleysolver: Online SMT algorithm selection. In: C. Li, F. Manyà (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings, *Lecture Notes in Computer Science*, vol. 12831, pp. 453–470. Springer (2021). DOI 10.1007/978-3-030-80223-3_31. URL https://doi.org/10.1007/978-3-030-80223-3_31
67. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: C. Bessiere (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4741, pp. 574–589. Springer (2007). DOI 10.1007/978-3-540-74970-7_41
68. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976). DOI 10.1016/S0065-2458(08)60520-3
69. Richter, C., Wehrheim, H.: Pesco: Predicting sequential combinations of verifiers - (competition contribution). In: D. Beyer, M. Huisman, F. Kordon, B. Steffen (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III, *Lecture Notes in Computer Science*, vol. 11429, pp. 229–233. Springer (2019). DOI 10.1007/978-3-030-17502-3_19
70. Rintanen, J.: Madagascar: Scalable planning with sat. Proceedings of the 8th International Planning Competition (IPC-2014) **21** (2014)
71. Rodríguez, J.D., Martínez, A.P., Lozano, J.A.: Sensitivity analysis of k-fold cross validation in prediction error estimation. IEEE Trans. Pattern Anal. Mach. Intell. **32**(3), 569–575 (2010). DOI 10.1109/TPAMI.2009.187
72. Salvia, R., Titolo, L., Feliú, M.A., Moscato, M.M., Muñoz, C.A., Rakamaric, Z.: A mixed real and floating-point solver. In: J.M. Badger, K.Y. Rozier (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings, *Lecture Notes in Computer Science*, vol. 11460, pp. 363–370. Springer (2019). DOI 10.1007/978-3-030-20652-9_25
73. Scott, J., Niemetz, A., Preiner, M., Nejati, S., Ganesh, V.: Machsmt: A machine learning-based algorithm selector for SMT solvers. In: J.F. Groote, K.G. Larsen (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 12652, pp. 303–325. Springer (2021). DOI 10.1007/978-3-030-72013-1_16. URL https://doi.org/10.1007/978-3-030-72013-1_16
74. Scott, J., Panju, M., Ganesh, V.: LGML: logic guided machine learning (student abstract). In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, pp. 13909–13910. AAAI Press (2020). URL https://aaai.org/ojs/index.php/AAAI/article/view/7227
75. Scott, J., Poupart, P., Ganesh, V.: An algorithm selection approach for QF_FP solvers. In: 17th International Workshop on Satisfiability Modulo Theories (2019)
76. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: S. Demri, D. Kapur, C. Weidenbach (eds.) Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8562, pp. 367–373. Springer (2014). DOI 10.1007/978-3-319-08587-6_28
77. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017). DOI 10.1007/s10817-017-9407-7
78. Tierney, K., Malitsky, Y.: An algorithm selection benchmark of the container pre-marshalling problem. In:

C. Dhaenens, L. Jourdan, M. Marmion (eds.) Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 8994, pp. 17–22. Springer (2015). DOI 10.1007/978-3-319-19084-6_2

79. Urban, J., Sutcliffe, G., Pudlák, P., Vyskocil, J.: Malarea SG1- machine learner for automated reasoning with semantic guidance. In: A. Armando, P. Baumgartner, G. Dowek (eds.) Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings, *Lecture Notes in Computer Science*, vol. 5195, pp. 441–456. Springer (2008). DOI 10.1007/978-3-540-71070-7_37
80. Vallati, M., Chrpa, L., Kitchin, D.E.: Portfolio-based planning: State of the art, common practice and open challenges. AI Commun. **28**(4), 717–733 (2015). DOI 10.3233/AIC-150671
81. Van Der Maaten, L., Postma, E., Van den Herik, J.: Dimensionality reduction: a comparative. J Mach Learn Res **10**(66-71), 13 (2009)
82. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019). DOI 10.3233/SAT190123. URL https://doi.org/10.3233/SAT190123
83. Wen, S.H., Mow, W.L., Chen, W.N., Wang, C.Y., Hsiao, H.C.: Enhancing symbolic execution by machine learning based solver selection (2019). DOI 10.14722/bar.2019.23080
84. Weston, J., Mukherjee, S., Chapelle, O., Pontil, M., Poggio, T.A., Vapnik, V.: Feature selection for svms. In: T.K. Leen, T.G. Dietterich, V. Tresp (eds.) Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA, pp. 668–674. MIT Press (2000). URL https://proceedings.neurips.cc/paper/2000/hash/8c3039bd5842dca3d944faab91447818-Abstract.html
85. Wold, S., Esbensen, K., Geladi, P.: Principal component analysis. Chemometrics and intelligent laboratory systems **2**(1-3), 37–52 (1987)
86. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In: C. Bessiere (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4741, pp. 712–727. Springer (2007). DOI 10.1007/978-3-540-74970-7_50
87. 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). DOI 10.1613/jair.2490
88. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla2009: an automatic algorithm portfolio for sat. SAT **4**, 53–55 (2009)
89. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: A. Cimatti, R. Sebastiani (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7317, pp. 228–241. Springer (2012). DOI 10.1007/978-3-642-31612-8_18
90. Xu, L., Hutter, F., Shen, J., Hoos, H.H., Leyton-Brown, K.: Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. Proceedings of SAT Challenge pp. 57–58 (2012)
91. Xu, R., II, D.C.W.: Survey of clustering algorithms. IEEE Trans. Neural Networks **16**(3), 645–678 (2005). DOI 10.1109/TNN.2005.845141. URL https://doi.org/10.1109/TNN.2005.845141