Machine learning for instance selection in SMT solving*

Daniel El Ouraoui*[1], Pascal Fontaine[1], and Cezary Kaliszyk[2]

[1] University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
{daniel.el-ouraoui,pascal.fontaine}@inria.fr
[2] University of Innsbruck, Innsbruck, Austria
cezary.kaliszyk@uibk.ac.at

Abstract. SMT solvers are among the most suited tools for quantifier-free first-order problems, and their support for quantified formulas has been improving in recent years. To instantiate quantifiers, they rely on heuristic techniques that generate thousands of instances, most of them useless. We propose to apply state-of-the-art machine learning techniques as classifiers for instances on top of the instantiation process. We show that such techniques can indeed decrease the number of generated useless instances and lead to more efficient SMT solving for quantified problems.

1 Introduction

Satisfiability modulo theories (SMT) solvers are among the best backends for verification tools and "hammers" in proof assistants. When proof obligations contain quantified formulas, SMT solvers rely on *instantiation*, replacing quantified subformulas by sets of ground instances. Three main instantiation techniques exist: enumerative [18], trigger-based [10], and conflict-based [19] instantiation. Among these, only conflict-based instantiation computes instances that are guaranteed to be relevant. It is however incomplete and has to be used in combination with other methods. Enumerative and trigger-based techniques are highly heuristic and generate a large number of instances, most of them useless. As a result, the search space of the solver explodes. Reducing the number of instances could improve the solver's efficiency and success rate within a given time limit.

We propose an instantiation technique based on machine learning. A predictor over the generated set of instances is trained on useful instances in previous runs of the SMT solver. That predictor is in turn used to filter out irrelevant instances, and thus decrease the number of instances given to the ground solver. The predictor is invoked after each instantiation round to evaluate the potential usefulness of each generated instance. Several strategies are then used to build a subset of potentially relevant instances that are immediately added to the ground solver. We conducted our experiments in veriT [6], an SMT solver that implements all three instantiation techniques described above. As the predictor, we chose the XGBoost gradient boosting toolkit [8] with the binary classification objective. This configuration has already been used successfully in the context of first-order theorem proving [13, 17].

In this paper we describe how such an approach can be integrated inside an SMT-solver to guide the instantiation process. Our experimental evaluation shows that the

number of instances is significantly decreased by this approach. We furthermore show that our prototype implementation leads to an increased amount of success on problems from the SMT-LIB. A preliminary version of this work was presented at the AITP workshop 2019 [5].

2 Preliminaries

The logic used in the context of SMT is many-sorted first-order logic with equality. We assume the reader to be familiar with the notions of *function*, *predicate*, *term*, (quantified and ground) *formula*, *literal*, *free variable*, and *substitution*. The notations $\overline{a_n}$ and $\overline{a}$ denote the tuple $(a_1, \ldots, a_n)$ with $n \geq 0$. We use the symbol $=$ for syntactic equality on terms and $\simeq$ denotes a family of a equality predicate, one for each sort. We reserve the names $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{p}$ for function symbols; x, y, z for variables in general; r, s, t, u for terms; and φ, ψ for formulas. The symbol $\models$ denotes the classical notion of entailment. The notation $t[\bar{x}_n]$ stands for a term whose free variables are included in the tuple of distinct variables $\bar{x}_n$; $t[\bar{s}_n]$ is the term obtained from t by a simultaneous substitution of $\bar{s}_n$ for $\bar{x}_n$. The symbol x denotes an integer (feature) vector, and notation $\mathsf{x}[i]$ stands for the ith element of the vector.

3 Instantiation

3.1 A short introduction to SMT

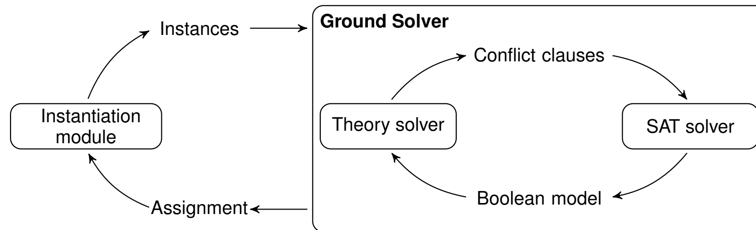

Fig. 1: The classic SMT core architecture.

The core of an SMT solver (Fig. 1) is a propositional satisfiability (SAT) solver [4], extended to more expressive logics with a theory solver (see [3] for more information about SMT). The input formula is abstracted to a Boolean formula, given to the SAT solver. The SAT solver provides a model for this Boolean abstraction, and the corresponding literals are checked for satisfiability by the theory solver. Thanks to this architecture, the theory solver only has to check the satisfiability of a conjunctive set of literals, and not an arbitrary Boolean combination. If the set of literals is unsatisfiable, the Boolean abstraction is refined through the addition of a propositional conflict clause to the SAT solver, and the process repeats. If the set of literals is satisfiable, then the

input formula is also satisfiable and a model can be output. If the theory is decidable, the theory solver always terminates, and if we further assume that conflict clauses only contain abstract Boolean variables from the input formula, the whole process eventually terminates, after the addition of a finite number of conflict clauses. If the formula is unsatisfiable, a proof can be provided.

The above process describes the internals of an SMT solver for a decidable theory and for quantifier-free problems. Quantified reasoning however requires additional techniques.

3.2 Instantiation in SMT

Quantifier reasoning is handled by an additional layer (see left hand side of Fig. 1), i.e., the instantiation module, on top of the ground SMT architecture described above. The ground SMT solver only sees a quantifier-free abstraction of the input, that is, the input where the quantified formulas are abstracted to new propositional formulas, after Skolemization. In a few words, the instantiation module in SMT is simply based on the foundational works of Skolem and Herbrand. The ground SMT core first produces a ground first-order assignment where quantified formulas appear as propositions. Then, the instantiation module provides instances of the quantified formulas with terms from the Herbrand universe (i.e., all possible well-sorted ground terms in the formula's signature). These instances will refine the ground abstraction of the formula, and the process repeats. Formally, given a set of literals or ground assignment E, and a set of quantified formulas Q, the *instantiation problem* is to find a set of instances I of Q such that $E \cup I \models \bot$. This is a semi-decidable problem in pure first-order logic with equality. Many approaches have been developed to tackle this problem: enumerative [18], trigger based [10] and conflict based instantiation [19].

Example 1. Consider the ground assignment $E = \{\neg\mathsf{P}(\mathsf{a}), \neg\mathsf{R}(\mathsf{b}), \mathsf{R}(\mathsf{a})\}$ and the quantified formula $\forall x.\, \mathsf{P}(x) \vee \neg\mathsf{R}(x)$. Instantiating the variable x with the Herbrand term a provides the ground conflicting instance $\mathsf{P}(\mathsf{a}) \vee \neg\mathsf{R}(\mathsf{a})$. •

Enumerative instantiation Thanks to the Herbrand Theorem, any quantified formula $\forall x \,.\, \psi[x]$ can be seen as an infinite conjunction $\bigwedge_t \psi[t]$ over all Herbrand terms t of appropriate sort; then the compactness Theorem states that there is always a finite unsatisfiable subset for any unsatisfiable set of formulas. Thus, for completeness, it is enough to blindly but fairly enumerate Herbrand instances to address the problem of instantiation.

Trigger based instantiation Rather than blindly instantiate using arbitrary Herbrand terms, this technique extracts sets of patterns (terms with free variables) from quantified formulas. These patterns, a.k.a. *triggers*, are matched with ground terms belonging to E, and the corresponding instances are generated. For instance, consider formula $\forall x.\; f(g(x)) \simeq x$. A suitable trigger is $f(g(x))$. With such a trigger, trigger based instantiation would generate an instance with $x \mapsto c$ when matched with $E = \{a \simeq f(b), b \simeq g(c)\}$. Several strategies have been developed to efficiently select and match triggers [1, 10, 11].

Conflicting based instantiation Reynolds et al. [19] introduced this technique to improve the performance of SMT solvers on quantified unsatisfiable problems. This approach repeatedly considers each quantified formula $\forall \overline{x}.\ \varphi$ in Q and tries to find a substitution σ for the set of variables $\overline{x}$ such that $E \models \neg\varphi\sigma$. The Congruence Closure with Free Variables (CCFV) algorithm is an improvement of this technique by Barbosa et al. [2].

The veriT solver [6] implements the three aforementioned instantiation methods. The following strategy is furthermore adopted:

i Try conflicting based instantiation;
ii if (i) fail, try trigger based instantiation;
iii if neither (i) and (ii) succeed, use enumerative based procedures.

The conflicting based procedures always produce good instances that contradict the assignment set E. On the other hand, it can only find conflicts with one clause at a time. In other words, if it is necessary to use two instances to contradict an assignment E, finding those two instances is beyond the scope of the method. Completeness requires using trigger and enumerative instantiation, but they are highly prolific and often lead the ground solver into a large search tree by adding many new instances and terms, some of them totally irrelevant.

3.3 The quest for good instances

Triggers and enumerative based instantiation are highly heuristic and generate a large number of instances. If the number of instances grows, the number of ground assignments and their size also grow. The task of instantiation may consequently be even more complicated by this amount of irrelevant information. Cutting irrelevant instances at the source might prevent this side effect, leading to improvements in efficiency and number of solved problems. Since machine learning has been successfully applied in many areas where a lot of data has to be sorted into relevant and irrelevant information, it seems natural to use it in this context.

Supervised learning needs a set of clearly labeled examples on which to learn on. To figure out if an instance is relevant — that is, if it is useful to solve the problem — the proof produced by the solver can be inspected *a posteriori*. The veriT solver produces fine grained proofs. It is easy to prune these proofs of the lemmas or instances that have not been used to deduce the unsatisfiability. By inspecting a proof, it is thus easy to sort out useful and useless instances.

By comparing the number of instances produced in a typical run of the solver and the ones appearing in the pruned proof, we get that only 10% of the produced instances are part of the pruned proof. In other words, on average, only 10% of the added formulas by the instantiation module are really useful. We want to train a machine learning classifier, to be used on top of the instantiation module to filter out the useless instances.

Let us illustrate this on a simple problem — a simplified version of the SMT-LIB benchmark `UF/misc/set10.smt2` — that will also serve as a running example.

Example 2. Consider the following set of axioms, where $\sqcup, \sqcap, \sqsubset, \in$ are uninterpreted symbols abstractly representing union, intersection, inclusion and membership predicates. For readability, infix notations is used.

$$\begin{aligned}
\varphi_1 &= \forall xyz \,.\, x \in y \wedge y \sqsubset z \Rightarrow x \in z \\
\varphi_2 &= \forall xy \,.\, \neg(x \sqsubset y) \Rightarrow \exists z \,.\, z \in x \wedge \neg(z \in y) \\
\varphi_3 &= \forall xyz \,.\, x \in (y \sqcup z) \simeq (x \in y \vee x \in z)
\end{aligned}$$

Round	E_i	φ_i	σ	instances
1	$\{\mathsf{a} \sqsubset \mathsf{c},\ \mathsf{b} \sqsubset \mathsf{c},\ \neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c})\}$	φ_2	$\sigma_{2,1}$	$x \mapsto \mathsf{a}, y \mapsto \mathsf{c}$
			$\sigma_{2,2}$	$x \mapsto \mathsf{b}, y \mapsto \mathsf{c}$
			$\sigma_{2,3}$	$x \mapsto (\mathsf{a} \sqcup \mathsf{b}), y \mapsto \mathsf{c}$
2	$E_1 \cup \{\mathsf{sk} \in (\mathsf{a} \sqcup \mathsf{b}),\ \neg(\mathsf{sk} \in \mathsf{c})\}$	φ_2	$\sigma_{2,4}$	$x \mapsto \mathsf{a}, y \mapsto \mathsf{c}$
			$\sigma_{2,5}$	$x \mapsto \mathsf{b}, y \mapsto \mathsf{c}$
			$\sigma_{2,6}$	$x \mapsto (\mathsf{a} \sqcup \mathsf{b}), y \mapsto \mathsf{c}$
		φ_3	$\sigma_{3,1}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{a}, z \mapsto \mathsf{b}$
3	$E_2 \cup \{\neg(\mathsf{sk} \in \mathsf{b}),\ \mathsf{sk} \in \mathsf{a}\}$	φ_1	$\sigma_{1,1}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{a}, z \mapsto \mathsf{c}$
4	$E_2 \cup \{\mathsf{sk} \in \mathsf{b},\ \neg(\mathsf{sk} \in \mathsf{a})\}$	φ_1	$\sigma_{1,2}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{b}, z \mapsto \mathsf{c}$

Table 1: Instantiation rounds of $E \cup Q$

Let us further assume a set of ground literals $E = \{\neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c}), \mathsf{a} \sqsubset \mathsf{c}, \mathsf{b} \sqsubset \mathsf{c}\}$, where $\mathsf{a}, \mathsf{b}, \mathsf{c}$ are constants symbols, and $Q = \{\varphi_1, \varphi_2, \varphi_3\}$. We want to solve the instantiation problem $E \cup I \models \bot$ where I is the set of ground instances of Q that should be generated. The set $E \cup Q$ is trivially unsatisfiable, but this is already an interesting case to analyze the general behavior of the solver. Table 1 enumerates the instantiation rounds a typical SMT solver would follow. The set E_i represents an assignment produced by the ground solver at the round i and φ_i the quantified formula from Q instantiated using substitutions in column σ. The instances with substitutions $\sigma_{1,1}$ and $\sigma_{1,2}$ are conflicting based, all others are either generated by trigger or enumeration based instantiation. The instances associated to φ_2 are formulas with one quantifier that is Skolemized. The constant sk comes from the Skolemization of φ_2 after instantiation with $\sigma_{2,3}$; for simplicity, we do not mention the other Skolem constants that are generated for the other (useless) instances. At the fourth instantiation round, the instance $\mathsf{sk} \in \mathsf{b} \wedge \mathsf{b} \sqsubset \mathsf{c} \Rightarrow \mathsf{sk} \in \mathsf{c}$ is generated, making the problem inconsistent at the ground level.

Looking at the instances column on the right, we can observe that some instances are redundant (e.g., $\sigma_{2,4}, \sigma_{2,5}, \sigma_{2,6}$) or useless (e.g., $\sigma_{2,1}, \sigma_{2,2}$) to solve the problem. A pruned proof would only contain instances $\sigma_{2,3}, \sigma_{3,1}, \sigma_{1,2}, \sigma_{1,1}$ (see Table 2). An ideal solver would thus only instantiate the formulas in Table 2, and would avoid swamping the ground solver with all the useless and redundant ones. It is easy to filter out redundant instances. We next show how to train an extreme gradient boosted tree to recognize and discard irrelevant instances. •

Round	E_i	φ_i	σ	instances
1	$\{\mathsf{a} \sqsubset \mathsf{c},\ \mathsf{b} \sqsubset \mathsf{c},\ \neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c})\}$	φ_2	$\sigma_{2,3}$	$x \mapsto (\mathsf{a} \sqcup \mathsf{b}), y \mapsto \mathsf{c}$
2	$E_1 \cup \{\mathsf{sk} \in (\mathsf{a} \sqcup \mathsf{b}),\ \neg \mathsf{sk} \in \mathsf{c}\}$	φ_3	$\sigma_{3,1}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{a}, z \mapsto \mathsf{b}$
3	$E_2 \cup \{\neg(\mathsf{sk} \in \mathsf{b}),\ \mathsf{sk} \in \mathsf{a}\}$	φ_1	$\sigma_{1,1}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{a}, z \mapsto \mathsf{c}$
4	$E_2 \cup \{\mathsf{sk} \in \mathsf{b},\ \neg(\mathsf{sk} \in \mathsf{a})\}$	φ_1	$\sigma_{1,2}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{b}, z \mapsto \mathsf{c}$

Table 2: Pruned instantiation rounds of $E \cup Q$

4 A machine learning approach for instantiation

Machine learning is progressively making its way into automated reasoning, notably for first-order theorem proving [9, 12]. This work is, to our best knowledge, the first to use machine learning for instantiation in SMT. To cope with this new application, machine learning algorithms need to tackle a few challenges. First, the training data is very imbalanced, since there are ten times more irrelevant instances than relevant ones. Second, the machine learning method will have to learn from a small number of examples — around one hundred thousands of examples extracted from a small set of problems — the training set being reduced to the problems available in the SMT-LIB. Third, the prediction should be inexpensive, so that it can be quickly applied to the large number of generated instances.

Among the many available supervised machine learning algorithms, state-of-the-art boosted decision trees meet these three requirements. We opted for the XGBoost [8] "eXtreme Gradient Boosting tree" toolkit used with the binary classification objective, which already proved helpful in [13, 17]. XGBoost is a highly scalable supervised machine learning algorithm with an open source implementation. We describe here how XGBoost is trained in order to guide the instance selection inside our SMT solver veriT.

4.1 From SMT to machine learning

Traditional machine learning algorithms work on *features*, i.e., numeric values that characterize the inputs. The role of the learning algorithm is essentially to identify and classify regions of this space of features. The features are the crucial elements linking the application to the machine learning algorithm. Applying such machine learning algorithms to new applications thus boils down to adequately representing the knowledge of the application using vectors of features in a Euclidean space and finding appropriate parameters for the algoritms. The goal of the features is to provide a representation that is as faithful as possible, to link the original application to the input of the machine learning technique. Being faithful is not always possible, due to many technical reasons. In that case, it is necessary to approximate the problem by featuring meaningful quantities for the application, e.g., in our context, depth, size of terms, used symbols.

4.2 Designing features

Features have been considered in the context of theorem proving in some previous work [15], in particular in combination with gradient boosted trees [17]. Our features

are more specifically inspired from the ones used in ENIGMA [13] and RLCoP [14]. This section presents the encoding for terms, formulas and assignments into a feature vector space, to be processed by the XGBoost algorithm.

The ground assignment in SMT solvers is a set of ground literals. Each term is associated to a feature vector. Although the solver internally represents terms as directed acyclic graphs, we use the tree representation for feature computation. Variables are abstracted away by a placeholder constant ($\circledast$ in the following), following [13]. Similarly Skolem constants are replaced by the placeholder $\odot$. For each sequence of symbols from root to a leaf, we extract all directed paths of length one, two and three. We refer to these paths as term walks. Sequences represent features and number of occurrences are the values. For instance, the term $\mathsf{f}(\mathsf{a},\mathsf{b})$ results in the set of features $\{\mathsf{f},\mathsf{a},\mathsf{b},(\mathsf{f},\mathsf{a}),(\mathsf{f},\mathsf{b})\}$. The value for the feature f is 3 since f appears three times in the path walks, whereas the value for (f,a) is 1.

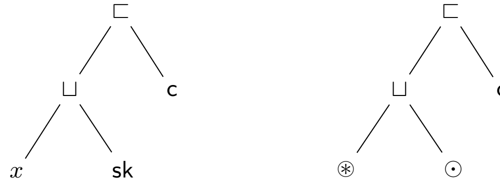

Fig. 2: Tree representing the literal $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$

Example 3. Figure 2 shows the tree representation of the term $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$ where x is a variable and sk a Skolem symbol. The left-hand side tree is the original term and the right-hand side tree is the processed tree, after replacing variables and Skolem constants by their respective placeholders.

Example 4. Consider the term $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$ from the previous example. The following table shows the extracted features from this term with their value. The k row gives the length of the term walks. The value is the number of times the sequence of symbols appear in all term walks of length at most 3.

k	1	value	2	value	3	value
`features`	$\sqsubset$	5	$(\sqsubset,\sqcup)$	2	$(\sqsubset,\sqcup,\circledast)$	1
	$\sqcup$	3	$(\sqsubset,\mathsf{c})$	1	$(\sqsubset,\sqcup,\odot)$	1
	$\odot$	3	$(\sqcup,\circledast)$	1		
	$\circledast$	2	$(\sqcup,\odot)$	1		

4.3 Problem description

In order to transform such tuples of symbols into positions in the feature vector, we use Vowpal Wabbit hashing [16]. Given the above characterization of terms by features,

we will now use it in the statement of a machine learning problem inside the SMT solver. The input of the predictor is a feature vector containing three parts (with disjoint features), for the quantified formula $\psi[\overline{x_n}]$, the substitution $\overline{x_n} \mapsto \overline{t_n}$ generating the instance of the quantified formula, and the ground assignment, i.e., a set of literals $l_1, \ldots, l_m$. For terms occurring in practice most features will be zero, since they do not appear in the term walks. Therefore sparse vectors are suitable for feature representation.

The encoding of the state is fairly complete, but in practice, it is also too expensive memory-wise. In order to decrease the amount of information, we do not use the full quantified formula $\psi[\overline{x_n}]$, but the triggers associated with the quantified formula. Furthermore, rather than taking into account all terms in $l_1, \ldots, l_m$, we only select the terms that are equal or disequal, according to $l_1, \ldots, l_m$, to one of the terms $t_1, \ldots, t_n$.This new state description allows us to generate a much smaller data set and considerably reduce the size of the trees used in the trained XGBoost prediction models. With the previous description, trees of depth 45 were necessary for decent results. With the reduced description, equivalently good results are obtained with trees of depth 20. On average the reduced description decreases the size of the generated samples by a factor between 50 and 100. As a side effect, much smaller training times are required.

The feature vectors also contain some extra abstract features such as size, depth, number of Skolem constants, number of terms and number of triggers. More precisely, for the terms in the substitution, for triggers, and for terms that are equal or disequal to terms in the substitution, we also add their depth and size. For terms in the substitution as well as for terms that are equal or disequal to them, there is a feature providing the number of terms equal to them according to $l_1, \ldots, l_m$. Finally, there are features informing about the total number of triggers, the total number of Skolem constants, and the maximal depth of the terms in the substitution.

Example 5. Consider the instance $\sigma_{2,1}$ from Example 2. The quantified formula is φ_2, and the substitution is $\sigma_{2,1}$ as represented by Table 3. There are no equalities in E_2. Thus, there are no other terms equal to a or c: the feature vector corresponding to the literals only contains features for a and c, and the feature vector is represented by Table 4. Notice that Table 3 and Table 4 differ by the ids of the features, since these ids for the substitution part and for the literals part are disjoint. The sole trigger in φ_2 is $\neg(x \sqsubset y)$ as represented by Table 5. The feature vector thus contains all Tables 3–5, as well as the abstract features mentioned above.

•

Example 6. To illustrate the features corresponding to literals, consider now the second round of Example 2 above, with literals $\mathsf{a} \simeq \mathsf{sk}$ and $\mathsf{b} \not\simeq \mathsf{a}$ now added to E_2. The terms equal to substituted terms yield the features in Table 7 because of the equality $\mathsf{a} \simeq \mathsf{sk}$. Furthermore, there is now one disequality which produces the features in Table 7. The trigger table is unchanged.

•

n	features	value
1	a	1
2	c	1

Table 3: The substitution

n	features	value
3	a	1
4	c	1

Table 4: The equal terms

n	features	value
5	$\circledast$	4
6	$\neg$	3
7	$\sqsubset$	4
8	$(\neg, \sqsubset)$	2
9	$(\sqsubset, \circledast)$	2
10	$(\neg, \sqsubset, \circledast)$	2

Table 5: The triggers

n	features	value
11	a	1
12	b	1

Table 6: The disequal terms

n	features	value
3	a	2
4	c	1
13	sk	1

Table 7: The equal terms

4.4 The right place and role for the predictor

The previous section explained how to turn the SMT instantiation problem into a machine learning problem, this section explains how to fit in the predictor inside the solver.

A random forest [7] is a set of decision trees. Each internal node of the decision tree compares the value of a particular feature of the input to classify with a constant stored in that node. Each leaf node contains a numeric contribution to the decision. The output of the random forest applied to a sample is the sum of the values in the leaves corresponding to that sample. The construction of the forest based on a set of training examples is beyond the scope of this paper. XGBoost [8] allows for efficient construction and evaluation of such forests with good prediction quality. To further improve efficiency of the prediction, we choose to hard code the model (i.e., the forest of binary decision trees) produced by the XGBoost algorithm by compiling it into an efficient C program. Each tree of the original model is translated as an if-then-else construction.

Evaluating usefulness Essentially, each node in the tree based model is a numeric comparison between a feature value $\mathsf{x}[i]$ and a given constant found by the algorithm during the training. The computation for one tree is thus linear with respect to the maximal depth of the tree, hence at most linear with respect to the size of the tree. We typically choose 20 as the maximal depth of a tree. Thus, evaluating the prediction on such trees will involve computing as many comparisons. Each tree $\mathcal{M}_i$ returns a score $\mathcal{M}_i(\mathsf{x})$ which is a real value within the interval $(-\infty, \infty)$. For the full prediction the results associated to all trees need to be summed up. The sigmoid function to compute the prediction from a forest of n decision trees for the feature vector x is

$$\texttt{xgb_predict}(\mathsf{x}) = \frac{1}{1 + e^{-\sum_i \mathcal{M}_i(\mathsf{x})}}$$

which returns a real value between 0 and 1 where 0 stands for a useless instance and 1 for a useful instance.

Triggering off machine learning A key point to successfully use the predictor is to know its predicting limit. The XGBoost library associates a value to the *importance* of each feature in the computed model. Given a feature vector, some features will be present, and some will not. For instance, if a symbol does not appear neither in the literal part, nor in the triggers, nor in the substitution, its corresponding feature will not appear in the input of the predictor. It turns out experimentally that, if the values of the feature importance for the features in a vector corresponding to an instance are low, the prediction is of poor quality, and such instances should not be filtered out.

Given a feature vector x corresponding to an instance, we compute its importance $\texttt{importance}(\mathsf{x})$ as the average of all values of the importance of the features in the vector. If this value is below a certain parameter λ, the prediction as computed with $\texttt{xgb_predict}(\mathsf{x})$ is considered to be uninformed, and the instance is not filtered out. The prediction function that is actually used for filtering is

$$\texttt{predict}(\mathsf{x}) = Mi \begin{cases} \texttt{xgb_predict}(\mathsf{x}) & \text{if } \texttt{importance}(\mathsf{x}) \geq \lambda \\ 1 & \text{otherwise} \end{cases}$$

where λ is a fudge factor equal to 80 times the number of Skolem constants in the example experimentally determined to prodice good results.

Re-calibrating At each round of instantiation all produced instances are stored in a queue, together with their score, computed as described above. The instances with a score above 0.5 are added to the ground solver. The others are saved in memory, to be reevaluated at the next round. It might happen that the prediction assigns low scores to all instances. We observed that in that case, it is best to re-calibrate the selection filter, and run the selection again. In practice, it works well to rescue all instances with a score larger than

$$\texttt{rescue_value}(I) = \left| \texttt{mean}(I) - K\frac{\sigma(I)}{L} \right|$$

where $\texttt{mean}(I)$ and $\sigma(I)$ are respectively the mean and standard deviation of the set of scores of the instances, and K and L are two adjustable constants. In practice, we use $K = 0.26$ and $L = 10$ if $\texttt{mean}(I) < 0.1$, and 1 otherwise. Algorithm 1 gives an overview of the procedure. D is a global queue containing all instances that have been generated but not yet selected for addition to the ground solver. We add all the formulas if D is small (lines 2–3), otherwise the formulas with score above one half are added (lines 4–6). The condition $\texttt{irrelevant}(D)$ in the abstract algorithm 1 (line 2) is true if the previous instantiation round did not produce any new instances and none of the instances in D were selected. Re-calibration and rescuing of instances occur at lines 7–10. On line 11, all generated instances that have not been selected are added to D.

5 Evaluation

To evaluate the benefit of the techniques presented here, we first compare the gain in the number of instances necessary to solve the problem. Then we study the time

Algorithm 1: Instance selection

```
   Input: Q set of quantified formulas
   Output: S selected instances
 1 I = D
 2 if |D| < 100 ∨ irrelevant(D) then
 3     I = I ∪ TriggerEnum(Q)
 4 foreach φ ∈ I do
 5     if predict(features(φ)) > 0.5 then
 6         S = S ∪ {φ}
 7 if S = ∅ then
 8     foreach φ ∈ I do
 9         if predict(features(φ)) > rescue_value(I) then
10             S = S ∪ {φ}
11 D = I \ S
12 return S
```

and success rates with a few variants of our implementation. Experiments have been conducted in the SMT solver veriT, on machines with 2 Intel Xeon Gold 6130 with 16 cores/CPU and 192 GiB RAM. We ran our experiments using the benchmarks in the UF category of the 2017 edition of the SMT-LIB, counting 7562 formulas (732 are labeled as satisfiable, 3316 unsatisfiable, and 3514 unknown). All the information necessary to reproduce the experiments is available on the web page `https://members.loria.fr/DElOuraoui/smtml.html`.[3]

For all experiments we use a proof-producing version of veriT. In the learning phase, we compare the full output proof with the proof pruned of irrelevant instances to discriminate useful instances from unused ones. An instance produced in the run is tagged as useful if it occurs in the pruned proof. The problems that only require conflict based instances in their proofs are not used in training. Since we essentially want to filter out instances generated by trigger and enumerative based instantiation, we also ignore the (useful) instances generated by conflicting based instantiation (which amount to about 30% of the instances), for the remaining benchmarks. In the rest of the discussion here, we only consider numbers without conflicting instances. Around 90% of the remaining instances are useless. In previous work [5], we under-sampled the useless instances, in order to provide a training set with as many good instances as bad ones. Here, we rather used the `scale_pos_weight` parameter provided by the XGBoost library which automatically balances the data in the training process by using a different learning rate for the positive examples (we are thankful to Jan Jakubův and Josef Urban for recommending this parameter).

We first run veriT without learning assistance on all the formulas with a 60 seconds time limit. Filtering out the formulas that are solved using conflicting instantiation only, there remain 1865 formulas. This set is randomly divided into a training set (70% of

[3] We will use Zenodo for the final version.

the formulas) and a test set (30%). The version of veriT that uses the instance selection algorithm trained on the formulas by the first run will be referred to as veriT($\mathcal{M}$). Similarly, veriT($\mathcal{M}^2$) is the version of veriT that uses the instance selection algorithm with a model trained, this time, with the formulas solved by veriT($\mathcal{M}$). There are 1914 benchmarks used in the process. We also consider a portfolio strategy, veriT($\mathcal{M}+\mathcal{M}^2$), using successively, veriT without learning, veriT($\mathcal{M}^2$) and veriT($\mathcal{M}$) with one third of the timeout for each component.

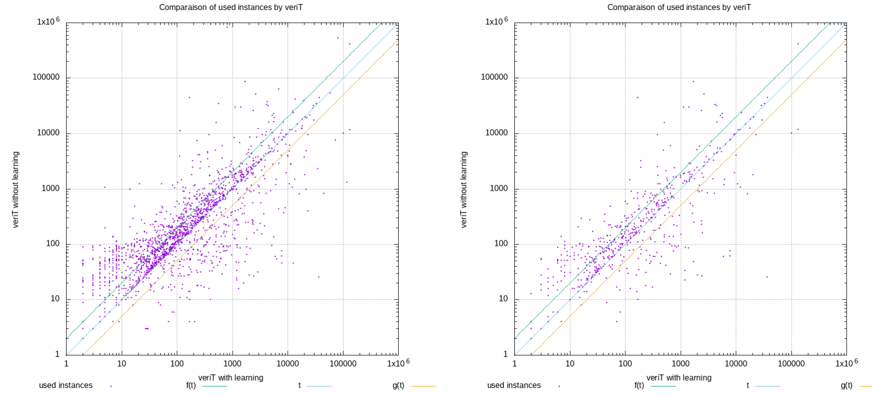

Fig. 3: Comparison of veriT configurations on UF SMT-LIB benchmarks

The numbers of generated instances for veriT (y-axis) and veriT($\mathcal{M}$) (x-axis) are compared in Figure 3. The fewer instances generated, the better the solver is. Keep in mind, however, that if a necessary instance is filtered out, the solver might be unable to prove the formula. The left-hand side shows the results of the solvers on the entire data set (training and test), whereas the right-hand side shows the results only on the test set. In both plots, a cluster of points is forming along the line corresponding to the equation $f(x) = 2x$. On average, instance selection allows the solver to find proofs with about half the number of instances, that is, the number of useless instances is decreased by a factor of two. These results show that our approach is suitable to substantially reduce the number of useless instances. We now show that it also allows the solver to prove more formulas.

Table 9 gives the number of problems solved within a given amount of time (30, 60, 120, and 180 seconds) for the different versions of veriT described above. As a base of comparison, we also provide the results using the state-of-the-art theorem prover Vampire. We used version 4.2.2 with the SMT-COMP mode option and without the Z3 SMT solver, since the experiments do not involve theory reasoning.[4] The conclusion is here that a trained solver with machine learning is better than the core solver. Comparison

[4] Note to the reviewers: our experiments with CVC4 exhibit results below Vampire. We plan to contact the CVC4 and Vampire teams so that the lines in Table 9 corresponding to those two mainstream provers are perfectly fair in the final version.

with other solvers is for information only; incidentally, using a trained SMT solver for the SMT-COMP competition would be dishonest.

Finally the rows labeled “portfolio” report on a portfolio strategy, using veriT with various configurations (as used in SMT-COMP), with and without the trained solvers in the portfolio. Learning on the problem successfully solved does indeed allow to solve more problems.

	30s	**60s**	**120s**	**180s**
veriT	2892	2906	2922	2927
veriT($\mathcal{M}$)	2904	2915	2925	2936
veriT($\mathcal{M}^2$)	2914	2927	2936	2942
veriT($\mathcal{M}+\mathcal{M}^2$)	2934	2957	2968	2971
veriT + portfolio	3176	3211	3226	3232
veriT($\mathcal{M}+\mathcal{M}^2$) + portfolio	3184	3240	**3307**	3317
Vampire smtcomp mode	**3274**	**3286**	3297	**3319**

Table 9: Results on the benchmarks in the UF category of the SMT-LIB.

Table 10 shows the gain and loss with respect to the number of solved problems. For instance, veriT($\mathcal{M}$) solves 40 problems that veriT does not solve, whereas veriT solves 32 problems that veriT($\mathcal{M}$) does not. Remember that the 40 problems are problems that have not been used to train veriT($\mathcal{M}$), since only the problems solved by veriT are used to train veriT($\mathcal{M}$). Using a portfolio is a way to mitigate the fact that tempering the instantiation algorithm will inevitably have negative effects for some formulas.

	veriT	veriT($\mathcal{M}$)	veriT($\mathcal{M}^2$)	veriT($\mathcal{M}+\mathcal{M}^2$)
veriT	0	32	27	5
veriT($\mathcal{M}$)	40	0	32	2
veriT($\mathcal{M}^2$)	46	43	0	7
veriT($\mathcal{M}+\mathcal{M}^2$)	66	55	49	0

Table 10: Comparative results on the UF SMT-LIB category, with 60 seconds timeout.

6 Conclusion

We have presented a method to combine instantiation techniques with machine learning for instance filtering in SMT solvers. A significant part of our feature representation uses elements of the syntax, which means that the technique would not work well when training on some set of problems and then using the solver on problems coming from a completely different area. However, our experiments demonstrate that machine learning

techniques can be used successfully to train a solver to improve itself on a coherent set of problems.

Future works involve improving the features: first experiments show that our way to compute the integers associated to features (using hashes) induces quite a lot of clashes, and fixing this leads to better results. Our features are currently quite dependent on the syntax of formulas: with features that are more independent of the syntax, training might prove advantageous also for formulas very different from the training set. Finally, we here investigated one kind of learning technique: other state-of-the-art techniques remain to be tried on this application.

Acknowledgement We are grateful to Jasmin Blanchette for many discussions throughout the development of this work, for providing funding for research visits and for suggesting many improvements. We also thank Hans-Jörg Schurr for suggesting many improvements. The work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka and No. 714034, SMART). Experiments were carried out using the Grid'5000 testbed (`https://www.grid5000.fr/`), supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations.

References

1. Kshitij Bansal, Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies. Deciding local theory extensions via e-matching. In Daniel Kroening and Corina S. Păsăreanu, editors, *CAV 2015*, LNCS. Springer International Publishing, 2015.
2. Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. Technical report, Inria, 2017. `https://hal.inria.fr/hal-01442691`.
3. Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, pages 825–885. IOS Press, 2009.
4. Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.
5. Jasmin Christian Blanchette, Daniel El Ouraoui, Pascal Fontaine, and Cezary Kaliszyk. Machine learning for instance selection in smt solving. In Thomas Hales, Cezary Kaliszyk, Ramana Kumar, Stephan Schulz, and Josef Urban, editors, *Conference on Artificial Intelligence and Theorem Proving (AITP 2019)*, 2019.
6. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *CADE–22*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.
7. Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.
8. Tianqi Chen and Carlos Guestrin. XGBoost: a scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *KDD 2016*, pages 785–794. ACM, 2016.
9. Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. *CoRR*, abs/1903.03182, 2019.

10. Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *CADE–21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
11. Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding decision procedures to SMT solvers using axioms with triggers. *J. Autom. Reasoning*, 56(4):387–457, 2016.
12. Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Een, François Chollet, and Josef Urban. DeepMath–deep sequence models for premise selection. In Daniel D. Lee, Masashi Sugiyama, Ulrike V. Luxburg, Isabelle Guyon, and Roman Garnett, editors, *NIPS 2016*, pages 2235–2243. Curran Associates, 2016.
13. Jan Jakubův and Josef Urban. ENIGMA: efficient learning–based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *CICM 2017*, volume 10383 of *LNCS*, pages 292–302. Springer, 2017.
14. Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olšák. Reinforcement learning of theorem proving. In Samy Bengio, Hanna Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *NeurIPS 2018*, pages 8835–8846. Curran Associates, 2018.
15. Cezary Kaliszyk, Josef Urban, and Jiri Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *IJCAI 2015*, pages 3084–3090. AAAI Press, 2015.
16. John Langford, Lihong Li, and Alex Strehl. Vowpal wabbit open source project. Technical report, Yahoo Research, 2007.
17. Bartosz Piotrowski and Josef Urban. ATPboost: Learning premise selection in binary setting with ATP feedback. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 566–574. Springer, 2018.
18. Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In Dirk Beyer and Marieke Huisman, editors, *TACAS 2018*, volume 10806 of *LNCS*, pages 112–131. Springer, 2018.
19. Andrew Reynolds, Cesare Tinelli, and Leonardo de Moura. Finding conflicting instances of quantified formulas in SMT. In Koen Claessen and Viktor Kuncak, editors, *FMCAD 2014*, pages 195–202. IEEE, 2014.