MACHINE LEARNING FOR INSTANCE SELECTION IN SMT SOLVING

DANIEL EL OURAOUI [a], JASMIN BLANCHETTE [b,c], PASCAL FONTAINE [d], AND CEZARY KALISZYK [e]

[a] CLEARSY, Paris, France
e-mail address: daniel.el-ouraoui@clearsy.com

[b] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
e-mail address: j.c.blanchette@vu.nl

[c] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
e-mail address: jasmin.blanchette@inria.fr

[d] University of Liège, Liège, Belgium
e-mail address: Pascal.Fontaine@uliege.be

[e] University of Innsbruck, Innsbruck, Austria
e-mail address: cezary.kaliszyk@uibk.ac.at

Abstract. Satisfiability modulo theories (SMT) solvers are powerful tools used to check specifications of critical systems and to discharge proof obligations in proof assistants. For many such applications, quantifiers are necessary to express the problems. SMT solvers often fail to find proofs when many quantifiers occur in the input problem. To support quantifiers, SMT solvers rely on instantiation, and use heuristic techniques to generate instances. Often, thousands of instances are generated, and since the vast majority of them are useless, they impede the solver.

In this article, we use machine learning to predict the usefulness of an instance to decrease the number of instances generated and processed by the SMT solver. To this end, we propose a meaningful way to characterize the state of an SMT solver, we collect instantiation learning data, and we integrate a predictor in the core of a modern SMT solver. This ultimately leads to more efficient SMT solving for quantified problems. To our best knowledge, this is the first use of machine learning for instance selection in the context of SMT.

1. Introduction

Formal verification is a mean to ensure safety of computer programs and complex systems such as in transport, hardware design, and energy. Verification methods rely heavily on mathematical and logical methods to formally reason about the behavior of these systems. In particular, satisfiability modulo theories (SMT) solvers are often used as backends to discharge the numerous formulas emerging from verification. SMT solvers are able to check the satisfiability of large logical formulas written in expressive languages containing uninterpreted symbols as well as interpreted operators for various theories, such as arithmetic symbols and data-structure handling operators.

Preprint submitted to
Logical Methods in Computer Science

SMT solvers particularly excel at dealing with ground (i.e., quantifier-free) formulas. When proof obligations contain quantified formulas, SMT solvers rely on *instantiation*, replacing quantified subformulas by sets of ground instances handled by the ground solver at the core of the SMT solver. Four main quantifier instantiation techniques have been developed: enumerative [RBF18], trigger-based [dMB07, DNS05], conflict-based [RTdM14], and model-based [GdM09]. Among these, only conflict-based instantiation computes instances that are guaranteed to be relevant (i.e., conflicting instances). It is, however, incomplete and must be used in combination with other methods. The three other techniques are highly heuristic and generate a large number of useless instances. As a result, the solver's search space explodes. Quickly finding the right instances—that is, reducing the number of useless instances given to the ground solver—is imperative if we want to substantially improve the solver's efficiency.

Machine learning techniques have already been successfully used to guide the search in automated reasoning, notably for premise selection in first-order automatic theorem provers [JU17, PU18]. Here we propose, for the first time, to use machine learning to improve quantifier instantiation within an SMT solver, veriT. A predictor is invoked after each instantiation round to evaluate the potential usefulness of each generated instance. Based on the prediction, instances are either given to the ground solver, delayed, or discarded. Since quantifier instantiation involves thousands of instances for problems that should be solved by the solver within a few seconds, the predictor needs to be fast. We opted for the XGBoost toolkit [CG16] with the binary classification objective. We propose and implemented features that are meaningful for SMT solving (Section 4.2). The instantiation problem must be encoded to apply machine learning to it (Section 4.3). For a successful integration of machine learning into the SMT instantiation procedure, it is necessary to fine-tune when, where, and to what extent it is used (Section 4.6).

We conducted our experiments in veriT [BdODF09] (Section 5). The solver implements the instantiation techniques mentioned above except model-based instantiation. We provide the solver including the machine learning aspect, with source code, under a permissive license.[1] Our experimental evaluation exhibits that the number of generated instances is substantially decreased using this approach. We furthermore show that our prototype implementation leads to an increased success rate on problems from the SMT-LIB (the reference library of SMT problems), allowing our prototype implementation in veriT to match in efficiency, after learning, the best provers according to recent SMT-COMP results. The approach is exclusively developed for unsatisfiable benchmarks. This means that the problems available for learning are essentially the unsatisfiable problems from the SMT-LIB that are already within the solving capability of the veriT SMT solver.

A preliminary version of this work was presented at the AITP workshop 2019 [BOFK19].

2. Background

2.1. **Notations.** The logic used in the context of SMT is first-order logic with equality. We assume the reader is familiar with the notions of function, predicate, term, (quantified and ground) formula, literal, free variable, and substitution. The notation $\bar{a}_n$ denotes the tuple

[1]The reviewers can access the file here `https://github.com/delouraoui/these/blob/main/veriTML.tar.xz`. A Zenodo repository with all software and data related to this document will be referred here in the final version of this article.

$(a_1, \ldots, a_n)$ with $n \geq 0$. We also write $\bar{a}$ when n is clear from the context. We use the symbol $=$ for syntactic equality on terms and $\simeq$ for the equality predicate. The names $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{p}$ are reserved for function symbols; P, R for predicate symbols; x, y, z for variables; r, s, t, u for terms; and φ, ψ for formulas. The symbol $\models$ denotes logical entailment. The notation $t[\bar{x}_n]$ stands for a term whose free variables are included in the tuple of distinct variables $\bar{x}_n$. Then $t[\bar{s}_n]$ is the ground term obtained by simultaneously substituting $\bar{s}_n$ for $\bar{x}_n$ in t. If x is a vector, $\mathsf{x}[i]$ stands for the ith element of the vector.

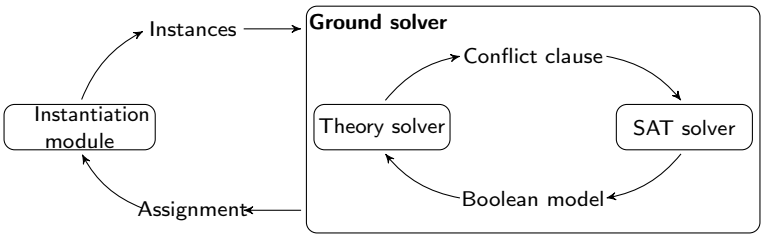

FIGURE 1. The classic SMT core architecture

2.2. **SMT.** A general introduction to SMT can be found in Barrett et al. [BSST09]. Briefly, the core of an SMT solver (Figure 1) is a propositional satisfiability (SAT) solver [BHvMW09] extended to more expressive logics with a theory solver. 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. This architecture allows the theory solver to check the satisfiability of a conjunctive set of literals rather than arbitrary Boolean combinations. 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 is repeated. 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 contain only 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.

Example 1. *Consider a quantifier-free first-order formula*

$$\mathsf{a} \simeq \mathsf{b} \wedge \big(\mathsf{f}(\mathsf{a}) \not\simeq \mathsf{f}(\mathsf{b}) \vee (\mathsf{R}(\mathsf{a}) \wedge \neg \mathsf{R}(\mathsf{b}))\big).$$

When given as input to an SMT solver, it is abstracted into the propositional formula

$$p_{\mathsf{a} \simeq \mathsf{b}} \wedge \big(\neg p_{\mathsf{f}(\mathsf{a}) \simeq \mathsf{f}(\mathsf{b})} \vee (p_{\mathsf{R}(\mathsf{a})} \wedge \neg p_{\mathsf{R}(\mathsf{b})})\big),$$

where p_ℓ denotes the Boolean abstraction of the atom ℓ. The embedded SAT solver might provide a first propositional model, satisfying $p_{\mathsf{a} \simeq \mathsf{b}}$ and $\neg p_{\mathsf{f}(\mathsf{a}) \simeq \mathsf{f}(\mathsf{b})}$. Since the set of literals $\{\mathsf{a} \simeq \mathsf{b}, \mathsf{f}(\mathsf{a}) \not\simeq \mathsf{f}(\mathsf{b})\}$ is unsatisfiable, the theory reasoner would then inform the SAT solver that this model is unacceptable by adding conjunctively to the original formula a conflict clause $\neg p_{\mathsf{a} \simeq \mathsf{b}} \vee p_{\mathsf{f}(\mathsf{a}) \simeq \mathsf{f}(\mathsf{b})}$. Next, the SAT solver would provide another propositional model, now satisfying $p_{\mathsf{a} \simeq \mathsf{b}}$, $p_{\mathsf{R}(\mathsf{a})}$, and $\neg p_{\mathsf{R}(\mathsf{b})}$, that would be refuted by the theory reasoner through the addition of yet another conflict clause $\neg p_{\mathsf{a} \simeq \mathsf{b}} \vee \neg p_{\mathsf{R}(\mathsf{a})} \vee p_{\mathsf{R}(\mathsf{b})}$. Finally, the SAT solver would conclude that the formula is unsatisfiable.

The above process describes the internals of an SMT solver for a decidable theory and for quantifier-free problems—that is, the ground SMT solver. If the ground formula is satisfiable, the ground solver comes up with a satisfiable conjunction of literals.

When formulas contain quantifiers, these quantified formulas are also abstracted as Boolean propositions. Dealing with them requires another feedback loop around the ground solver.

2.3. **Instantiation in SMT.** Quantifier reasoning is handled by an additional layer (on the left of Figure 1), called the *instantiation module*. It is built 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 fresh propositional formulas, after Skolemization.

The instantiation module in SMT is based on the foundational works of Skolem and Herbrand. Briefly, in refutation mode Skolemization makes it possible to remove in a satisfiability-preserving way all quantifiers with an existential meaning through the introduction of witnesses—i.e., fresh Skolem constants and functions. The other kind of quantifiers can be removed thanks to Herbrand theory, which states that a first-order Skolem formula is unsatisfiable if and only if there is a finite unsatisfiable conjunctive set of Herbrand instances. An *Herbrand instance* of a formula $\forall x_1 \dots x_n.\, \varphi[\bar{x}_n]$ is $\varphi[\bar{s}_n]$, where $\bar{s}_n$ is a tuple of arbitrary ground terms built using the symbols available in the formula. We refer to classical logic introductory books (e.g., Fitting [Fit90]) for a detailed presentation of Skolem and Herbrand theory.

In the context of SMT, the ground solver 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*, which consists of all possible ground terms in the formula's signature. These instances will refine the ground abstraction of the formula, and the process is repeated. Formally, given a set of ground literals E (the *ground assignment*) and a set of quantified formulas Q, the *instantiation problem* consists of finding a set of ground instances H of Q such that $E \cup H \models \bot$. This is a semidecidable problem in pure first-order logic with equality.

Example 2. *Consider the ground assignment* $E = \{\neg \mathsf{P}(\mathsf{a}), \neg \mathsf{R}(\mathsf{b}), \mathsf{R}(\mathsf{a})\}$ *and the quantified formula* $\varphi = \forall x.\, \mathsf{P}(x) \vee \neg \mathsf{R}(x)$*. The instantiation problem consists of finding a set of instances* H *of* φ *that is inconsistent together with* E*. Since the only terms in the Herbrand universe are* a *and* b*, Herbrand theory tells us that it is sufficient to consider* H *to comprise the two ground instances* $\mathsf{P}(\mathsf{a}) \vee \neg \mathsf{R}(\mathsf{a})$ *and* $\mathsf{P}(\mathsf{b}) \vee \neg \mathsf{R}(\mathsf{b})$*. Actually, the ground instance* $\mathsf{P}(\mathsf{a}) \vee \neg \mathsf{R}(\mathsf{a})$ *is conflicting. This sole instance together with* E *leads to a contradiction, whereas the other instance is useless.*

Many approaches have been developed to tackle the instantiation problem, including enumerative [RBF18], trigger-based [dMB07, DNS05], and conflict-based [RTdM14] instantiation. We will briefly describe them below, since they are relevant for our work and experimental evaluation. We will not discuss model-based quantifier instantiation [GdM09], since usually it can advantageously be replaced by a combination of the other approaches, especially if the objective is to refute formulas [RBF18].

Enumerative instantiation. Thanks to Herbrand theory, any quantified formula $\forall x.\, \psi[x]$ can be seen as an infinite conjunction $\bigwedge_t \psi[t]$ over all Herbrand terms t. Then the compactness theorem states that there is always a finite unsatisfiable subset for any unsatisfiable set of formulas. Thus, to get a complete method, it suffices to blindly but fairly enumerate Herbrand instances to address the problem of instantiation.

Trigger-based instantiation. Rather than blindly instantiating using arbitrary Herbrand terms, trigger-based instantiation extracts sets of patterns (terms with free variables) from quantified formulas. These patterns, called *triggers*, are matched with ground terms belonging to E, and the corresponding instances are generated. For example, consider the formula $\forall x.\, \mathsf{f}(\mathsf{g}(x)) \simeq x$. A suitable trigger is $\mathsf{f}(\mathsf{g}(x))$. With such a trigger, trigger-based instantiation would generate an instance with $x \mapsto \mathsf{c}$ when matched with $E = \{\mathsf{a} \simeq \mathsf{f}(\mathsf{b}), \mathsf{b} \simeq \mathsf{g}(\mathsf{c})\}$, since $\mathsf{f}(\mathsf{g}(c))$ implicitly belongs to the terms used in E. Several strategies have been developed to efficiently select and match triggers [dMB07, DNS05, DCKP16, BRK+15].

Conflict-based instantiation. Reynolds et al. [RTdM14] introduced conflict-based instantiation to improve the performance of SMT solvers on quantified unsatisfiable problems. This approach repeatedly considers each quantified formula $\forall \bar{x}.\, \varphi$ in Q and checks for the existence of a substitution σ for the set of variables $\bar{x}$ such that $E \models \neg\varphi\sigma$. These substitutions can be found using the congruence closure with free variables algorithm [BFR17]. In Example 2, the formula $\mathsf{P}(\mathsf{a}) \vee \neg\mathsf{R}(\mathsf{a})$ is a conflicting instance and would be obtained from E and φ using the algorithm.

The veriT solver implements the three instantiation methods described above. It first tries conflict-based instantiation; if this fails, it tries trigger-based instantiation; and if this also fails, it resorts to enumerative instantiation.

Conflict-based instantiation always produces useful 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 or more instances to contradict an assignment E, finding those instances is beyond the scope of the method. Completeness requires using trigger-based 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, most of them irrelevant.

3. The quest for relevant instances

Trigger-based 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 become even more complicated due to this amount of irrelevant information. Reducing the number of irrelevant instances at the source prevents this effect, leading to improvements in efficiency and increasing the number of solved problems. We investigate machine learning for this purpose.

Supervised learning needs a set of clearly labeled examples as training set. 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 analyzed a posteriori. The veriT solver produces fine-grained proofs. It is easy to prune these proofs of all the lemmas or generated instances that do not really play a role in deducing the unsatisfiability. By inspecting a proof, it is thus possible and inexpensive to determine which instances were useful. This is an approximation, because an instance

might not appear as essential in a proof, although there might exists another proof in which the same instance is relevant. The approximation has the benefit of being inexpensive.

Comparing the number of instances produced in a typical run of the solver and the ones appearing in the pruned proof, only around 10% of the produced instances are part of the pruned proof, and only 1%, are generated by conflict-based instantiation. This means that around as few as 10% of the instances added by the instantiation module are actually useful. A good classifier would eliminate some of the 90% of useless generated 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 3. *Consider the following set of axioms, where the* $\sqcup, \sqcap, \sqsubset, \in$ *are uninterpreted symbols abstractly representing union, intersection, inclusion, and membership, respectively, and are written in infix notation:*

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

Let us further fix 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 constant symbols, and a set of quantified formulas* $Q = \{\varphi_1, \varphi_2, \varphi_3\}$*. We want to solve the instantiation problem* $E \cup H \models \bot$*, where* H *is the set of ground instances of* Q *that must be generated.*

Table 1 enumerates the instantiation rounds that would take place in the veriT solver, leading to a contradiction. The second column provides the set of literals in an assignment produced by the ground solver at round i*, and the third column specifies the quantified formula from* Q *instantiated using the substitutions in the last column. The instances with substitutions* $\sigma_{1,1}$ *and* $\sigma_{1,2}$ *are conflict-based; all others are generated by either trigger or enumeration-based instantiation. The instances associated with* φ_2 *are formulas with one quantifier that is subsequently 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. Rounds 3 and 4 are induced by a branching of the solver on the literal* $\mathsf{sk} \in \mathsf{b}$*. As a result, at round 4, conflict-based instantiation produces the instance* $\mathsf{sk} \in \mathsf{b} \wedge \mathsf{b} \sqsubset \mathsf{c} \Rightarrow \mathsf{sk} \in \mathsf{c}$*, making the problem inconsistent at the ground level.*

TABLE 1. Instantiation rounds of $E \cup Q$

Round i	*Literals* E_i	*Formula*	*Substitution*
1	$\{\neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c}),\ \mathsf{a} \sqsubset \mathsf{c},\ \mathsf{b} \sqsubset \mathsf{c}\}$	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	$E_3 \cup \{\mathsf{sk} \in \mathsf{b},\ \neg(\mathsf{sk} \in \mathsf{a})\}$	φ_1	$\sigma_{1,2} = \{x \mapsto \mathsf{sk},\ y \mapsto \mathsf{b},\ z \mapsto \mathsf{c}\}$

Looking at the instances column on the right, we can observe that some instances are redundant ($\sigma_{2,4}$, $\sigma_{2,5}$, $\sigma_{2,6}$) or useless ($\sigma_{2,1}$, $\sigma_{2,2}$) to solve the problem. A pruned proof would contain only instances $\sigma_{2,3}$, $\sigma_{3,1}$, $\sigma_{1,2}$, and $\sigma_{1,1}$. An ideal instantiation module would thus only instantiate the formulas in Table 2.

While it is fairly easy to filter out redundant instances, simply by checking whether the instance has already been added to the ground solver, recognizing and discarding irrelevant instances is a difficult task. We next show how to train and use a machine learning model to detect irrelevant instances.

4. A learning approach to instantiation

To successfully use machine learning algorithms in our new context, we need to overcome a few difficulties. First, the training data is very unbalanced, 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 100 000 examples extracted from a small set of problems. Essentially, the training set is reduced to the problems available in the SMT-LIB repository of benchmarks. 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, boosted decision trees [CG16] seem to be particularly appropriate for the above context and have already proved helpful in other theorem proving settings [JU17, PU18]. We describe here how the predictor is trained to guide the instance selection inside veriT.

4.1. Encoding SMT as a classification problem.

Traditional machine learning algorithms work on *features*—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 represent the knowledge of the application using vectors of features in a Euclidean space and finding appropriate parameters for the algorithms. In other words, 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. For example, there is no discrete approach to faithfully encode first-order logic formulas. The issue is that there is no uniform representation that makes it possible to establish an equivalence between two formulas under renaming.

Ideally, to reduce overfitting, it might seem more appropriate to approximate the problem by taking abstract quantities such as the depth and size of terms or the used symbols as features, rather than the terms as they appear in the problem. Unfortunately, after multiple

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

Round i	*Literals E_i*	*Formula*	*Substitution*
1	$\{\neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c}), \mathsf{a} \sqsubset \mathsf{c}, \mathsf{b} \sqsubset \mathsf{c}\}$	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	$\sigma_{1,2} = \{x \mapsto \mathsf{sk}, y \mapsto \mathsf{b}, z \mapsto \mathsf{c}\}$

attempts, we could find no effective combination of such quantities during our experiments. Therefore, we have chosen to use a hybrid approach based mainly on syntactic encoding as well as on a combination of abstract quantities. In the following subsection, we develop this idea further and suggest an approach to encode the SMT formulas as integer vectors.

4.2. **Designing features.** The characterization of instantiation problems via feature vectors for the learning algorithm requires preliminary modeling work. The role of the learning algorithm is essentially to partition the feature space into regions. The features are the crucial elements linking the application—here, the instantiation problem in SMT—to the learning algorithm. Previous work [KUV15] in the context of automatic proof studied various types of features which have notably been used in combination with learning approaches based on decision trees [PU18].

Although effective, these approaches cannot directly be used for SMT solvers, since they are mainly developed for prover architectures based on saturation systems, such as tableaux and superposition. Because the architecture of SMT solvers is quite far from these solvers, it is necessary to rethink the modeling of the problem, while taking inspiration from previous studies. The features developed in this work are more specifically inspired from those used in ENIGMA [JU17] and rlCoP [KUMO18]. This section presents the encoding of terms, formulas, and satisfiable instances of the original problem into a feature vector space, to be processed by the classifier.

An SMT solver such as veriT works on terms of first-order logic, which are internally represented as directed acyclic graphs (dags). This representation saves memory and simplifies some algorithms. Duplicated subterms share the same memory space. In short, dags are a kind of compressed representation of abstract syntax trees.

The instantiation problem is essentially expressed with terms of first-order logic. Thus, to inform the learning algorithm for SMT instantiation, it is necessary to represent those terms in an appropriate way for the learning algorithm. This step is not trivial, since it essentially means dags representing terms must be encoded into integer vectors of a priori fixed size. In this representation, the particular function and predicate symbols used in terms will in some way play an important role.

Learning methods based on syntactic term encoding appear to be effective when problem benchmarks use a domain of *coherent* symbols. That is, the problem family sticks to a set of common symbols, and there are no (or very few) multiple definitions of the same notions with various symbols in a problem family. This is the case for formulas stemming from proof assistants such as Isabelle and Mizar. Since this is our main target application domain, we believe sensitivity to renaming is an acceptable drawback of our approach.

Our encoding is based on sets of sequences of symbols extracted from each term in the input problem. These symbol sequences correspond to the symbols met in the subtraversals from root to leafs of the syntax tree representation of terms. It appears experimentally that the best results are obtained by computing the feature vectors based on *sequences of lengths one, two, and three*. This experimental observation corroborates the works of Jakubův and Urban [JU17], who use sequences of length three only.

Each of the obtained sequences represents a feature, and the number of occurrences of a sequence is the value associated with the feature. The classifier might consider a sequence appearing several times as more important than a sequence appearing just once. For example, encoding the term $\mathsf{f}(\mathsf{a},\mathsf{b})$ results in the feature set $\mathsf{f}, \mathsf{a}, \mathsf{b}, (\mathsf{f},\mathsf{a}), (\mathsf{f},\mathsf{b})$.

FIGURE 2. Tree representation of the literal $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$

FIGURE 3. Tree representation of the literal $\mathsf{g}\,(\mathsf{f}\,x)\,(\mathsf{f}\,y)$

The names of variable and Skolem symbols are not really meaningful. Therefore, our encoding abstracts variables with a specific $\circledast$ symbol and Skolem symbols with $\odot$. The examples below illustrate the encoding for terms of first-order logic.

Example 4. *Figure 2 shows the tree representation of the term $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$ where x is a variable and sk is a Skolem symbol. The tree on the left is the original term, and the tree on the right is the processed tree, after replacing variables and Skolem constants by their respective placeholders.*

Table 3 shows the extracted features from this term together with their value. The first column provides the length of the term traversals. The value is the number of times the sequence of symbols appears in all subtraversals of length at most three.

Example 5. *Figure 3 presents the tree representing the term $\mathsf{g}\,(\mathsf{f}\,x)\,(\mathsf{f}\,y)$, before and after processing. Table 4 lists the features and their values.*

TABLE 3. Features of the tree of Figure 2

Length	*Feature*	*Value*
1	$\sqsubset$	5
	$\sqcup$	3
	$\odot$	3
	$\circledast$	2
2	$(\sqsubset, \sqcup)$	2
	$(\sqsubset, \mathsf{c})$	1
	$(\sqcup, \circledast)$	1
	$(\sqcup, \odot)$	1
3	$(\sqsubset, \sqcup, \circledast)$	1
	$(\sqsubset, \sqcup, \odot)$	1

The size of the feature vectors is fixed; our experience has shown that vectors with a dimension around 2 MB constitute a good compromise. This size allows us to store enough features for the problems encountered in our experiments, this parameter can be reevaluated for different problems.

To transform each sequence of symbols—that is, each feature—into an index into the feature vector, we use the `djb2` hash function:

```
Int64 djb2(char *str)
{
    unsigned long hash = 5381;
    int c;
    while (c = *str++)
        hash = ((hash << 5) + hash) + c;
    return hash;
}
```

The function allows us to associate with each symbol a natural integer. The transformation of symbol sequences into features is calculated by the recursive formula

$$u_{n+1} = u_n * k^{n+1} + s_n,$$

where n ranges from 0 to 2, u_n the prefix of u_{n+1} of length n, k is a large prime number, and s_n is the hash value of the nth symbol of the sequence. In the case of a sequence f_0 of length 1, the calculated feature corresponds to the hash value of the symbol modulo the size of the vector. For a sequence (f_0, f_1) of length 2, we calculate the feature as follows: $u_1 = \texttt{hash}(f_0) * k + \texttt{hash}(f_1)$. And the calculation of a sequence (f_0, f_1, f_2) of length 3 is $u_1 * k^2 + \texttt{hash}(f_2)$. Once the calculation is finished, each feature is brought back in the dimension of the vectors space by applying the modulo operator.

In practice, the features space is segmented so that features from, for example, the set of ground literals E, the instances, or abstract quantities (e.g., size and depths of terms) are distinguished from each other. For this purpose, a simple offset is used when calculating the features. For example if the sequence u_1 is extracted from the set E of equations, we add a fixed offset, d_1, multiplied by a certain interval: $u_1 + d_1 * 262139$ PF2DEO: is that important besides the fact that it is a large integer? DEO2PF: Yes to reduce clash with encoding. Greater is the integer more sequences you can encode x PF2CK: This is a prime

TABLE 4. Features of the tree of Figure 3

Length	*Feature*	*Value*
1	$\sqsubset$	5
	$\sqcup$	3
	$\odot$	3
	$\circledast$	2
2	$(\sqsubset, \sqcup)$	2
	$(\sqsubset, \mathsf{c})$	1
	$(\sqcup, \circledast)$	1
	$(\sqcup, \odot)$	1
3	$(\sqsubset, \sqcup, \circledast)$	1
	$(\sqsubset, \sqcup, \odot)$	1

number, again why is that a good choice? Otherwise, if the sequence comes from an instance, then the offset value will be different. This trick allows us to partition each vector so that the machine learning algorithm can identify the source of the information. In the next section we look in more detail at the modeling of the problem of instantiation in SMT. PF2DEO, PF2CK: this section is a mystery to me, there are too many magical numbers and it is not explained what is important about those numbers (e.g. large prime, near 2^{31},...). The mod stuff seemed applied at the wrong places. I do not understand why multiplying by a large prime number is useful. Edit: Daniel and I simplified a bit, but I am still not totally at ease with it. Cesary if you see how we can purify, please tell us.

4.3. **Problem description.** The features described in the previous section form the basis of the machine learning approach for the SMT instantiation problem. The input of the instantiation module is essentially a ground model—i.e., a set E of literals that satisfy the ground part of the formula and a set Q of quantified formulas. The instantiation module considers the various quantified formulas $\psi[\bar{x}_n]$ in Q, and possibly computes some substitution $\sigma = \bar{x}_n \mapsto \bar{t}_n$ corresponding to a generated instance $\psi[\bar{x}_n]\sigma = \psi[\bar{t}_n]$. For the learning algorithm, the relevant feature vector is

$$(\texttt{features}(E), \texttt{features}(\psi[\bar{x}_n]), \texttt{features}(\bar{x}_n \mapsto \bar{t}_n))$$

where the function `features` translates into a vector of features the relevant input elements for the instantiation module as well as the result. The core idea here is to inform the learning algorithm of the input and the result of each instantiation task, as well as whether it was fruitful.

In practice, each feature vector has a fixed length, usually a large natural number (Section 4.2). Most vectors computed for an instance contain only a small subset of symbol sequences. A sparse vector would seem to be an ideal representation, because only the nonzero values of the vector are stored, in a lossless way. Although this representation rather faithfully translates the state of the solver for an instantiation task, it appears in practice to still be too expensive. Too much information is recorded, and as a consequence, large models are produced by the learning algorithm. Because of these large models, the learning phase is very expensive, and even worse, the prediction is also expensive.

To reduce the amount of information, we only consider the triggers (Section 2.3) associated with the quantified formula instead of the whole quantified formula $\psi[\bar{x}_n]$. Thus, in Example 3, instead of considering the entire formula $\varphi_2 = \forall xy.\ \neg(x \sqsubset y) \Rightarrow \exists z.\ z \in x \wedge z \notin y$, only its trigger $\neg(x \sqsubset y)$ is considered. Note that even for enumerative instantiation, we use the triggers as suitable extracts of formulas for the purpose of computing features. Moreover, rather than taking into account all the terms in E, we select only one element per congruence class according to E. In other words, if two terms t_1 and t_2 are such that $E \models t_1 \simeq t_2$, only one is selected, namely, the representative of the congruence class in the congruence closure algorithm. This new representation of the state allows us to associate with each instance produced by the instantiation module a vector of features of considerably reduced size and that encapsulates the essential information to perform quantifier instantiation.

To improve the classifier predictions, some amount of abstract information about the structure of the terms is added to each feature vector. More specifically, for each state, we compute the size, the average and maximum depth of the terms appearing in the substitution, the number of Skolem constants, the total number of terms, the number of triggers, the average depth of the triggers and the terms related to the substitution as well as the sum of

TABLE 5. The features for the substitution in Example 6

ID	*Feature*	*Value*
1	a	*1*
2	c	*1*

TABLE 6. The features for the equal terms in Example 6

ID	*Feature*	*Value*
3	a	*1*
4	c	*1*

TABLE 7. The features for the triggers in Example 6

ID	*Feature*	*Value*
5	$\circledast$	*2*
6	$\neg$	*1*
7	$\sqsubset$	*1*
8	$(\neg, \sqsubset)$	*2*
9	$(\sqsubset, \circledast)$	*2*
10	$(\neg, \sqsubset, \circledast)$	*2*

the sizes of the equivalence classes in E of each term of the substitution. The example below illustrates concretely the characterization of the state in the feature vector.

Example 6. *Consider the instance $\sigma_{2,1}$ from Example 3. The quantified formula is φ_2, and the substitution is $\sigma_{2,1}$ represented by Table 5. There are no equalities in E_1. Thus, there are no other terms equal to* a *or* c *according to E_1. The feature vector corresponding to the literals, presented in Table 6, contains only features for* a *and* c*. PF2DEO: I do not understand that. There are many terms in E_2... DEO2PF: The only term you can use for instanciation here are the terms in table 4... PF2DEO: Why so? Nothing prevents you to use $a \sqcup b$, that also appears in E_1, right? DEO2PF: Ok, after reading again the paper, I think I remember what I wanted to say. I wanted to say that there is only "a" single in its equivalent class and "c" single in its class. Then there are only these two terms that can be chosen modulo equality? But for example, if we had $a = c$, for example, we would have only one value "a" with value 2 (as in example 7) in the table 4. Is it clearer now...? ... PF2DEO: we are getting close: why don't we have $a \sqcup b$ as a term somewhere Notice that Tables 5 and 6 differ by the feature identifiers, since the IDs for the substitution part and for the literals part are disjoint (Section 4.2). The only trigger in φ_2 is $\neg(x \sqsubset y)$, shown in Table 7. The feature vector is thus made up of all Tables 5 to 7, together with the abstract features mentioned above.*

Example 7. *To illustrate the features corresponding to literals, consider now the second round of Example 3, but assume that literals* a $\simeq$ sk *and* b $\not\simeq$ a *also belong to E_2. The*

TABLE 8. The features of the disequal terms in Example 7

ID	*Feature*	*Value*
11	a	*1*
12	b	*1*

TABLE 9. The features of the equal terms in Example 7

ID	*Feature*	*Value*
3	a	*2*
4	c	*1*
13	$\odot$	*1*

terms equal to substituted terms yield the features in Table 9 because of the equality $\mathsf{a} \simeq \mathsf{sk}$. *Furthermore, there is now one disequality which produces the features in Table 8. The trigger table is as in the previous example.*

4.4. **Machine learning.** Machine learning algorithms are tools based on mathematical and statistical approaches. Initially developed to solve a wide range of problems from observations, these approaches make it possible to learn tasks without having any real prior knowledge of the application domain. The machine learning process generally comprises two phases. The first one is called the learning phase, when a predictive model is built from a finite set of observations. In our context, this consists of collecting examples of instances as defined in the previous section, annotate them as useful or useless, and then train the algorithm to build a predictive model from these observations. The second phase is the evaluation or prediction phase, when the predictive model built in the learning phase is used as a classifier to evaluate new observations.

Predictions can be of two kinds. We may want to use the machine learning algorithm as a classification engine, where each observation must be assigned to a class. In the training phase, observations are given together with the class they actually belong to. Alternatively, we may want the machine learning algorithm to predict the value of some function associated with the observation. In the training phase, observations are given together with the value of the function.

Depending on the available data, some learning methods are more suitable than others. If the set of annotated observations is sufficiently large, it is possible to use a supervised learning approach. The most popular learning methods that use this principle are the support vector machine approach [CL11, Vap00], the k nearest neighbor method, artificial neural network systems (also called deep learning when they are composed of several sublayers) [LBH15], decision trees [DP09], and boosting [FSA99]. We settled for boosted decision trees [CG16].

Another popular method in machine learning is reinforcement learning. This approach is based on a reward system. Each choice made by the algorithm is evaluated by a reward function, which judges the right choices positively. This approach is unsupervised—i.e., no prior observation is necessary. The system is embedded in the program, or the application, and is designed to learn the "best" decisions dynamically, throughout its execution. A very well-known example of this approach is AlphaGo [SSS+17], known for having beaten in May 2017 the go champion Ke Jie after playing a large number of virtual games. This approach has also been successfully integrated into the leanCoP [OB03] automatic prover, resulting in the rlCoP [KUMO18] prover which, after several uses, starts overperforming the nonlearning prover.

Thus, whatever the objective, or the type of application of the machine learning method, it is essential to have a large number of different observations. The better the examples are, the more accurate and efficient the produced model will be to perform predictions on new observations. A predictive model is said to generalize the problem if it gives correct answers based on a wide range of new observations. On the contrary, we want to avoid overfitting, a phenomenon that tends to occur when the training set of observations is too small or too specific. In case of overfitting, the obtained model is very good on the train set of observations, but the predictions on new observations are poor.

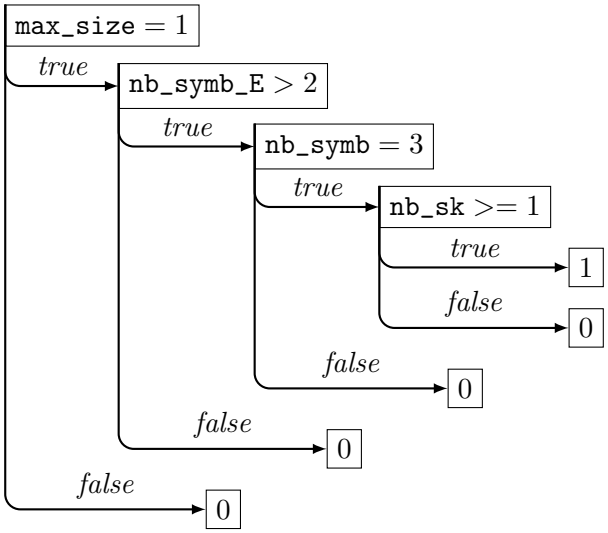

FIGURE 4. A decision tree

4.5. **Decision trees.** In this article, we use a supervised learning approach, based on decision trees. More precisely we use the XGBoost [CG16] algorithm, which is a relatively sophisticated approach combining two learning methods: decision trees and boosting. Moreover, the learning process is improved by a gradient descent approach to optimize the search for predictive models. We provide here, without going into details, an intuition on the internals of this algorithm.

Overall, decision trees are simple probabilistic tools that allow us to make a choice according to several factors. The trees are composed of nodes, branches, and leaves. Each node describes the distribution of the random variable to be predicted PF2ALL: I am not at all easy with "Each node of the tree describes the distribution of the random variable to be predicted", but I need somebody more comfortable with AI and stats to fix this. Cezary?, each branch corresponds to a choice, and each leaf corresponds to a value for the prediction. PF2ALL: I do not understand this sentence, I suggest removing: There are as many branches as there are values to predict. The tree is then built according to the observations in the training set. Here, we want to predict if a particular instance is useful or not. For this, we build a binary tree, whose depth depends on the number of features collected in the observations.

Example 8. *To predict an instance usefulness, we could for example rely on the maximum size* `max_size` *of the terms obtained in the substitution, the number* `nb_symb_E` *of symbols that appear in* E*, the number* `nb_symb` *of different symbols, and the number* `nb_sk` *of Skolem symbols. These features are simplistic but will help us explain how decision trees work. Assume we have the decision tree in Figure 4, and we want to predict the usefulness of instance* $\varphi_2\sigma_{2,1}$ *of Example 3. We compute the features for* $\varphi_2\sigma_{2,1}$ *and get* `max_size` $= 1$, `nb_symb_E` $= 2$, `nb_symb` $= 2$, *PF2DEO: is it??? DEO2PF: no you right, nb_symb the number of ground symbols is 3 {a, b, c} PF2DEO: I am still confused: in nb_symb_E, nb_symb don't you count also the nary symbols like* $\sqcup$ *and* `nb_sk` $= 0$. *Following the path in the tree in Figure 4, we reach a leaf labeled by* 0 *and conclude that the instance* $\varphi_2\sigma_{2,1}$ *is not useful.*

The particularity of the XGBoost algorithm is that it does not build a single tree but several. This is the boosting principle, which is based on the following hypothesis: A set of weak classifiers gives us a strong classifier. A weak classifier is a classifier that alone is not capable of producing usable predictions; by contrast, a strong classifier is a classifier capable of interpreting the entire feature domain of the input problem. Generally, each classifier is specialized on a restricted subdomain of features. However, when several of these classifiers are combined, the boosting principle states that the interpretation of the sum of these predictions is more efficient than that of a single strong classifier. In practice, the prediction is thus obtained through a voting system between the classifiers. In the XGBoost algorithm, each weak classifier is a decision tree.

The XGBoost algorithm relies on a gradient descent to optimize each tree produced during the training phase, making the predictions of each new tree forest increasingly accurate while learning. For more details, we refer to the original XGBoost paper [CG16] and to the seminal paper on random forests [Bre01].

4.6. **Integrating the predictor into veriT.** The previous subsection explains how the XGBoost learning algorithm works. In this subsection, we will focus on the integration of the algorithm into veriT.

Evaluating usefulness. The predictive model produced by the XGBoost algorithm is a collection of decision trees. The accuracy of the predictions depends on two parameters: the first is the depth of each of these trees, and the second is the number of trees. The distribution of features for each of the trees is random, meaning that a single tree does not contain all the features of the problem; however, the algorithm distributes the features so that the tree forest can evaluate any input containing the features seen during the learning phase.

Ideally, to minimize prediction times, it would be desirable to train models with few and shallow trees in proportion to the total number of features contained in our set of observations. In our case, the constraint on the prediction time is strong. Indeed, since we want to evaluate the usefulness of each instance, it is important to not delay the solver during this filtering phase. In Section 5, we will determine empirically what number of trees and what maximum tree depth to use.

Given a feature vector x corresponding to an instance, each tree $\mathcal{M}_k$ returns a score $\mathcal{M}_k(\mathsf{x})$, which is a real value in the interval $(-\infty, \infty)$. To process the predictions of the predictive model, we must first sum the set of values returned by each tree, then pass this value in a nonlinear "activation" function, which lets us bring this sum in a real interval $(0, 1)$. The sigmoid function for computing the prediction from a forest of n decision trees for the feature vector x is as follows:

$$\texttt{xgb\_predict}(\mathsf{x}) = \frac{1}{1 \ + \ e^{-\sum_k \mathcal{M}_k(\mathsf{x})}}.$$

It yields a real value between 0 and 1, 0 corresponding to a seemingly useless instance, and 1 to a seemingly useful instance. It is this function which is used to evaluate each feature vector.

Leveraging predictions. We have described the mechanisms to evaluate an instance through a feature vector. It is now important to understand how to leverage these predictions to help the solver solve new problems.

To fully exploit the capabilities of the classifier, we must design a system able to determine whether the symbols present in the instance we want to evaluate are symbols known by the classifier. To go further, it is possible to classify all the features appearing in the model by order of importance. A feature is more important than another if it appears more often and higher in the model's trees. XGBoost lets us extract from a model an association table of value of *importance* for each feature of the model. Thus, on a new observation, the feature vector computed from the method described above will contain a number, possibly null, of features present in the model. Using the association table generated from the model, we can compute for each collected observation an importance value. This value can then be used to evaluate the relevance of each prediction. Thus, if the importance of the features of a vector corresponding to an observation is low, the prediction might be of poor quality and will be considered uninformative.

Given a feature vector $\mathtt{x}$ corresponding to an instance, we compute its importance $\mathtt{imp}(\mathtt{x})$ as the average of all feature importance values in $\mathtt{x}$. If this value is less than a certain parameter λ, the prediction computed by $\mathtt{xg\_predict}(\mathtt{x})$ is considered uninformative, and the instance is not filtered. The prediction function that is used for filtering is the following:

$$\mathtt{predict}(\mathtt{x}) = \begin{cases} \mathtt{xgb\_predict}(\mathtt{x}) & \text{if } \mathtt{imp}(\mathtt{x}) \geq \lambda \\ 1 & \text{otherwise} \end{cases}$$

where the factor λ depends on the number of Skolem symbols present in the instance, more precisely 80 times the number of Skolem symbols. This value has been determined experimentally and gives good results in practice.

Instance selection. We will now present the approach implemented in veriT to filter the instances produced by the instantiation module, generated from the strategies by trigger or by enumeration. At each instantiation cycle, all the produced instances are stored in a queue, as well as their score, calculated as described above. Instances with a score higher than 0.5 are added to the ground solver. The others are kept, to be reevaluated in the next round. In some cases, the predictions may be low for all instances. Therefore, if the filtering limit value is not reevaluated, the solver may simply abandon the problem due to a lack of instances. We have observed that in these cases it is better to recalibrate the selection filter and restart the selection. Thus, in practice, when the average score of the instances is too low, a new filtering value is computed, and instances are reconsidered for addition to the ground solver against this new value. This is called recovery, or rescue of instances. In practice, the recovery value is computed as follows:

$$\mathtt{rescue\_value}(H) = \left| \mathtt{mean}(H) - 0.26\frac{\sigma(H)}{10} \right|,$$

where $\mathtt{mean}(H)$ and $\sigma(H)$ are respectively the mean and standard deviation of the set of scores of the instances, and 0.26/10 is an adjustable value that appears to work well in practice.

Algorithm 1 gives an overview of the procedure. H is a global queue containing all the instances that have been generated in previous cycles but have not yet been selected for addition to the ground solver. Each new set of instances is generated using the $\mathtt{TriggerEnum}(Q)$

function, which implements the instantiation module using the triggers and enumeration based strategies.

If the H queue is small (lines 2 and 3 of Algorithm 1), or if instances are not relevant, the algorithm asks the instantiation module to produce new instances via a call to $\texttt{TriggerEnum}(Q)$. More precisely, the condition $\texttt{irrelevant}(H)$ in Algorithm 1 (line 2) is true if the instantiation cycle has not produced any new instances, and none of the instances of H have been selected. Otherwise, it means that the instances generated in the previous cycle can be used, and it is not necessary to generate new ones yet. The algorithm will then try to filter the instances. First, the algorithm evaluates the score of each instance (lines 4 to 6), using the function $\texttt{predict}(\texttt{features}(\varphi))$, where $\texttt{features}(\varphi)$ computes the vector of features for an instance φ. If an instance scores above 0.5, the algorithm stores the instance in S for future addition to the ground solver. Finally, if no instance has been selected by the previous filtering pass on lines 4 to 6, the algorithm triggers the rescue process (lines 7 to 10), which filters the instances with a weaker condition $\texttt{rescue\_value}(H)$ that is necessarily smaller than 0.5. On line 10, H is cleaned of the selected instances.

Input: Q set of quantified formulas
Output: S selected instances
1 $S = \emptyset$
2 **if** $|H| < 100 \vee \texttt{irrelevant}(H)$ **then**
3 $H = H \cup \texttt{TriggerEnum}(Q)$
4 **foreach** $\varphi \in H$ **do**
5 **if** $\texttt{predict}(\texttt{features}(\varphi)) > 0.5$ **then**
6 $S = S \cup \{\varphi\}$
7 **if** $S = \emptyset$ **then**
8 **foreach** $\varphi \in H$ **do**
9 **if** $\texttt{predict}(\texttt{features}(\varphi)) > \texttt{rescue\_value}(H)$ **then**
10 $S = S \cup \{\varphi\}$
11 $H = H \setminus S$
12 **return** S

Algorithm 1: Instance selection

In this section, we have presented the instance selection algorithm used in veriT to evaluate each of the instances produced by the triggers-based and enumeration-based instantiation strategies. In the next section, we will present the results obtained with the implementation presented above.

5. Evaluation

To evaluate the benefit of the techniques presented here, we first compare the number of instances necessary to solve the problem with and without instance selection. Then we study the time and success rates with several 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 a recent edition of the SMT-LIB, PF2DEO: Is there a chance to update this? consisting of 7572 problems: 771 labeled as satisfiable, 3442 labeled as unsatisfiable, and

3359 labeled as unknown. All the information necessary to reproduce the experiments is available on the web page `https://members.loria.fr/DElOuraoui/smtml.html`.[2]

For the 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 useless ones. An instance produced in the run is tagged as useful if it occurs in the pruned proof. The problems that require only conflict-based instances in their proofs are not used in training. Since we want to filter out instances generated by trigger and enumeration-based instantiation, we also ignore the useful instances generated by conflict-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. Rather than oversample the useful instances, we use a higher learning rate to balance the dataset. (We are thankful to Jan Jakubův and Josef Urban for recommending this.)

We first run veriT without learning assistance on all the problems with a 60 seconds time limit. Filtering out the problems that are solved using conflicting instantiation only, there remain 1865 problems. We randomly divided this into a training set consisting of 70% of the problems and a test set consisting of the remaining 30%.

The presented approach is based on pattern (symbol sequences) frequency occurrence of appearing more recurrently in a problem. Such an approach may seem weak at first, but can be very useful in the context of computer-aided proof, especially when used as support for the user. Indeed, when working on formal proofs, via tools such as Isabelle, Coq, or Atelier B, the set of symbols is generally quite limited: The same set of predicates, functions, and constants appear in many proof obligations. The user has to solve a large number of proofs, by regularly calling automatic solvers such as SMT solvers. We therefore imagine here a system capable of integrating with this type of tool, and capable of getting hints or observations from the easy proof obligations to tackle the more complicated ones. When the number of observations is large enough to produce a robust model, the model is used by the SMT solver as a filter to select good instances when checking the most challenging proof obligations. Incrementally, any new observations can be added to the knowledge base to produce a new model more robust than the previous one, so that the solver becomes more and more efficient on new problems.

The evaluations presented in this section aim to simulate this behavior. We incrementally run two models. A first model is produced from a set of observations extracted via the problems that the SMT solver is able to solve without instance filtering. In a separate step, a second model is produced from observations extracted from solved problems from a version of the solver that uses instance filtering, and whose predictions are based on those of the first model. The version of veriT that uses the instance selection algorithm trained on the problems 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 with the problems solved by veriT($\mathcal{M}$). There are 1914 benchmarks used in the process. PF2DEO: why 1914? I am confused with the above number of 1865. DEO2PF: this is the exact partition of the training set and test set used in our exps JB2DEO: I'm also confused. 0.3 or 0.7 * 1865 $\neq$ 1914. And I'm confused by "in the process": do you mean for learning (0.7) or for evaluation (0.3)? Please help us converge here. We also consider a portfolio strategy veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$),

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

which runs veriT, then veriT($\mathcal{M}^2$), and last veriT($\mathcal{M}$), each time with one third of the time limit.

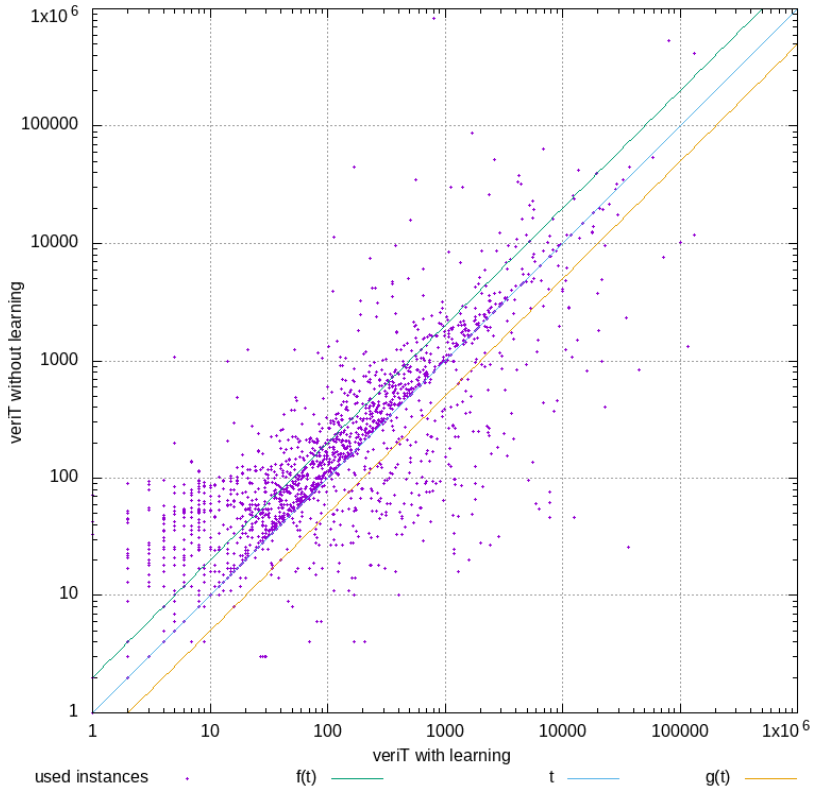

FIGURE 5. Comparison of the numbers of generated instances by the veriT configurations with and without learning on the 1914 benchmarks

Figure 5 compares veriT and veriT($\mathcal{M}$) on the UF SMT-LIB benchmarks (the 1914 benchmarks). A point in the Figure 5 is read, along the x-axis, as the number of generated instances by veriT($\mathcal{M}$), and, along the y-axis, as the number of instances generated by veriT. The fewer instances generated, the better we consider the solver. Keep in mind, however, that if a necessary instance is filtered out, the solver might be unable to prove the problem.

The figure shows the results of the solvers on the entire data set (training and test). In the plot, a cluster of points is forming along the line corresponding to the equation $f(x) = 2x$. This means that learning saves about half of the instantiations on average. The comparison on the test set only, in Figure 6, produces a plot comparable to Figure 5 but with a fewer points.

These results suggest 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 problems. We first present the results obtained on the UF category benchmarks, by subcategories. In these comparisons, only the UF subcategories containing a sufficient number of problems are presented in the tables below. All these tables compare the number of unsatisfiable problems solved, in a given time limit, for the different versions of the veriT SMT solver.

We compare these different versions of veriT to the results obtained by the two solvers that obtained the best results, before veriT, at the 2019 and 2020 SMT-COMP, in this

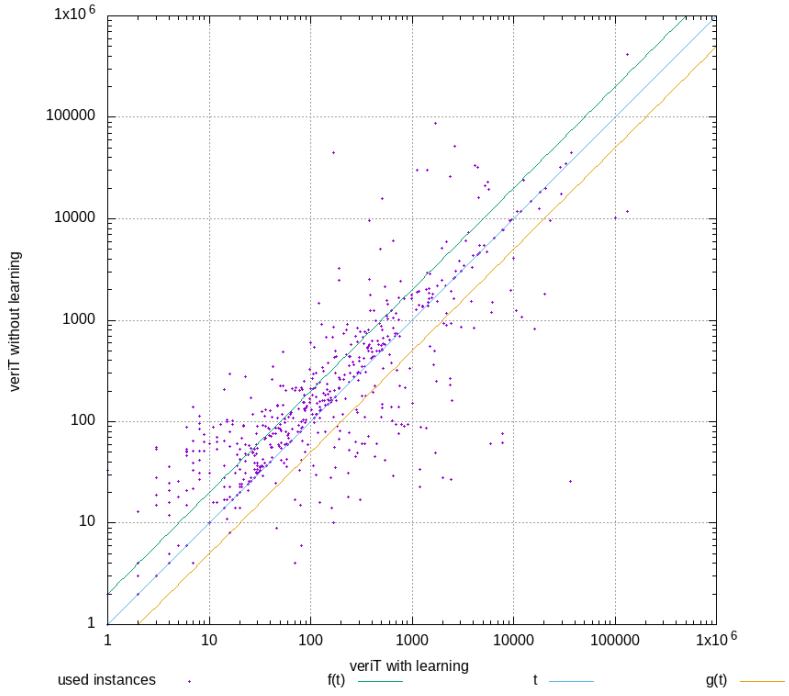

FIGURE 6. Comparison of the numbers of generated instances by the veriT configurations with and without learning on the test set only

TABLE 10. Results on the problem set without the problems used to train veriT($\mathcal{M}$)

	Sledgehammer (3675 problems)	Grass (204 problems)	Barrett (2371 problems)	Total (6250 problems)
veriT	609	196	822	1627
veriT($\mathcal{M}$)	612	195	829	1636

category, which are respectively the SMT solver CVC4 [DRK+14] and the automatic theorem prover Vampire [KV13]. We use version 4.2.2 of Vampire with the dedicated option used for the SMT-COMP, and without help of Z3 since the evaluations do not involve theory reasoning. Finally, the lines labeled "portfolio" show a portfolio strategy, using veriT with various configurations (like those used in SMT-COMP), with and without the learning strategies. CVC4 and Vampire use a portfolio approach: The execution of many different strategies on the same problem in a fraction of the time gives generally much better results than running a single configuration for the entire time.

The problems used to train the models are removed from the benchmarks used for the evaluation. Thus Table 10 above compares veriT to veriT($\mathcal{M}$) under a time limit of 180 seconds on the UF benchmarks without the problems used to train the model used by veriT($\mathcal{M}$). Table 11 does the same for veriT and veriT($\mathcal{M}^2$).

Table 12 investigates a portfolio approach including veriT($\mathcal{M}$) and veriT($\mathcal{M}^2$), again under a time limit of 180 seconds. For this evaluation, the problems used to train the veriT($\mathcal{M}$) model and the veriT($\mathcal{M}^2$) model have been removed. This configuration is the

TABLE 11. Results on the problem set without the problems used to train veriT($\mathcal{M}^2$)

	Sledgehammer (3669 problems)	Grass (192 problems)	Barrett (2379 problems)	Total (6240 problems)
veriT	609	183	836	1628
veriT($\mathcal{M}^2$)	614	184	842	1640

TABLE 12. Results on the problem set without the problems used to train veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$)

	Sledgehammer (3529 problems)	Grass (122 problems)	Barrett (2180 problems)	Total (5831 problems)
veriT	470	114	638	1222
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$)	482	116	652	1250
veriT + portfolio	667	121	727	1515
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$) + portfolio	721	121	752	1595

TABLE 13. Results on the benchmarks in the UF category of the SMT-LIB

	30 s	60 s	120 s	180 s
veriT	2896	2913	2923	2929
veriT($\mathcal{M}$)	2907	2917	2925	2936
veriT($\mathcal{M}^2$)	2916	2927	2935	2944
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$)	2936	2959	2969	2975
veriT + portfolio	3181	3215	3228	3234
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$) + portfolio	3190	3247	3312	3322
Vampire smtcomp mode	3154	3165	3175	3197
CVC4 portfolio	3311	3345	3393	3404

one that exhibits the best results. We can also observe that veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$) can solve 28 more problems than veriT. Even better results are observed with the portfolio version of veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$), which solves 80 more problems than the regular portfolio approach used at the SMT-COMP competition.

Tables 13 and 14 show the results obtained on all the problems of the UF category. For these evaluations, no problems have been removed. Table 13 gives a comparison of all the configurations of veriT with the two solvers CVC4 and Vampire for various time limits: 30, 60, 120, and 180 seconds.

Since our target application is mainly proof assistants, it should also produce good results under short time limits. Table 15 uses a time limit of JB2DEO: How many seconds? 60? seconds and provides separate numbers for different benchmark categories. In particular, we notice the substantial positive influence of machine learning for instance filtering for the Sledgehammer category, which best represents our target application.

The conclusion of these evaluations is that a solver using a learning approach can solve problems more efficiently than classical SMT solvers, especially with a high time limit. The evaluation with the other solvers is only informative. Moreover, using an SMT solver with this

TABLE 14. Results on the benchmarks in the UF subcategory of the SMT-LIB

	Sledgehammer	Misc	Grass	Barrett
veriT	1072	14	431	1412
veriT($\mathcal{M}$)	1073	14	431	1418
veriT($\mathcal{M}^2$)	1079	14	431	1420
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$)	1093	14	434	1434
veriT + portfolio	1276	14	439	1505
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$) + portfolio	1333	14	440	1535
Vampire smtcomp mode	1389	15	436	1481
CVC4 portfolio	1366	14	440	1584

TABLE 15. Comparative results on the SMT-LIB UF category, with a 60 s time limit

	veriT	veriT($\mathcal{M}$)	veriT($\mathcal{M}^2$)	veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$)
veriT	0	39	32	15
veriT($\mathcal{M}$)	47	0	33	1
veriT($\mathcal{M}^2$)	48	41	0	10
veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$)	64	42	44	0

type of approach for the SMT-COMP competition would be unfair, since the trained solver somehow remembers the formulas it has already seen, and takes shortcuts in its search space, selecting the right instances. The point is rather here that an SMT solver with a machine learning method for instance selection can learn from previous successes and consequently solve even more problems. Within a moderately efficient solver, the learning approach would somewhat compensate for the absence of a large number of carefully hand-tuned heuristics and strategies, such as those implemented in state-of-the-art solvers.

Table 15 shows the gain and loss with respect to the number of solved problems for the versions veriT, veriT($\mathcal{M}$), veriT($\mathcal{M}^2$), and veriT($\emptyset$+$\mathcal{M}$+$\mathcal{M}^2$). Each numeric table cell compares two solver configurations: It provides the number of problems lost by the configuration associated with the column compared with the configuration associated with the row. Alternatively, a cell can also be read as the number of problems won by the configuration of the row compared with the configuration of the column. For example, veriT($\mathcal{M}$) solves 47 problems that veriT does not solve, whereas veriT solves 39 problems that veriT($\mathcal{M}$) does not solve. Bear in mind that the 39 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 compensate for the loss of some problems by the instantiation algorithm, which inevitably has a negative impact for some problems.

6. Related work

Machine learning has been considered to guide several other aspects of automatic provers and proof assistants. Premise selection [AHK+14] or relevance filtering [BGK+16] is the problem of selecting a reasonably small subset of a larger lemma base to pass to an automatic prover to prove a given conjecture. Various machine learning techniques including gradient

boosted trees [PU18] and deep learning [ISA+16] have been tried for this problem, and some provers include relevance filtering procedures. Machine learning has also been applied to predict the satisfiability and equivalence of expressions [ACKS17] and even to directly synthesize proofs in simpler logics [SS18]. Finally, machine learning has been used to directly guide automatic theorem proving procedures. This was first done for the selection of extension steps in tableau provers [UVv11], more recently combined with Monte Carlo proof search [KUMO18]. For the superposition calculus, the selection of next clauses to process has been tried in the E prover [LISK17], including learned watchlist guidance [GJU19]. Actual synthesis of substitutions using deep learning approaches has been tried in the Holophrasm prover [Wha16]; however, only very small useful terms could be generated. To our knowledge, none of the existing approaches considers learning useful instances.

7. Conclusion

When proving quantified formulas, SMT solvers use instantiation techniques that create a lot of instances, and only small proportion of them are useful. The useless instances hurt the efficiency of the solver. We have here presented a method to combine instantiation techniques with machine learning for instance filtering. Our experiments demonstrate that machine learning techniques can be used successfully to train an SMT solver to improve itself on a coherent set of problems. We believe that our methods are helpful for an SMT solver to learn from its success, when used as a backend of an application generating many formulas involving the same concepts. This is typically the case for verification applications. Learning from the easy problems can then help tackling the more challenging ones.

As future work, we could improve the characterization of the state of an SMT solver. This can be done both by improving the features or by relying on a neural network to find a better representation automatically. Our first experiments show that our way to compute the integers associated with features (using hashes) leads to many clashes, and addressing this might lead to better results. We also plan to investigate more syntax-independent features, which would also help an SMT solver transfer knowledge learned from one set of problems to other problems stemming from different areas. Other learning techniques could be tried as well. Finally, selecting good instances is also an important problem in more sophisticated logics—e.g., higher-order logic.

Acknowledgments. We thank Hans-Jörg Schurr for his comments. 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. We are thankful to the reviewers of a preliminary version of this work [BOFK19] for their comments.

References

[ACKS17] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles A. Sutton. Learning continuous semantic representations of symbolic expressions. In Doina Precup and Yee Whye Teh, editors, *ICML 2017*, volume 70, pages 80–88. PMLR, 2017.

[AHK+14] Jesse Alama, Tom Heskes, Daniel Kühlwein, Tsivtsivadze Evgeni, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2), 2014. doi:10.1007/s10817-013-9286-5.

[BdODF09] 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. doi:10.1007/978-3-642-02959-2_12.

[BFR17] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. Technical report, Inria, 2017. URL: https://hal.inria.fr/hal-01442691.

[BGK+16] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016. doi:10.1007/s10817-016-9362-8.

[BHvMW09] 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, 2009.

[BOFK19] 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, *AITP 2019*, 2019.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 2001. doi:10.1023/A:1010933404324.

[BRK+15] 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*, volume 9207 of *LNCS*. Springer, 2015. doi:10.1007/978-3-319-21668-3_6.

[BSST09] 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*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi:10.3233/FAIA201017.

[CG16] 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. doi:10.1145/2939672.2939785.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. doi:10.1145/1961189.1961199.

[DCKP16] 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. doi:10.1007/s10817-015-9352-2.

[dMB07] 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. doi:10.1007/978-3-540-73595-3_13.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. doi:10.1145/1066100.1066102.

[DP09] Steinberg Dan and Colla Phillip. CART: Classification and regression trees. In Xindong Wu and Vipin Kumar, editors, *The Top Ten Algorithms in Data Mining*, volume 9 of *Data Mining and Knowledge Discovery Series*, page 179. CRC Press, 2009.

[DRK+14] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: How it works, and how to use it. In *FMCAD 2014*. IEEE, 2014. doi:10.1109/FMCAD.2014.6987586.

[Fit90] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, Berlin, 1990.

[FSA99] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *J. JSAI*, 14(771-780):1612, 1999.

[GdM09] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009. doi:10.1007/978-3-642-02658-4_25.

[GJU19] Zarathustra Goertzel, Jan Jakubuv, and Josef Urban. ENIGMAWatch: ProofWatch meets ENIGMA. In Serenella Cerrito and Andrei Popescu, editors, *TABLEAUX 2019*, volume 11714 of *LNCS*, pages 374–388. Springer, 2019. doi:10.1007/978-3-030-29026-9_21.

[ISA+16] 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.

[JU17] 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. doi:10.1007/978-3-319-62075-6_20.

[KUMO18] 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.

[KUV15] 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.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*. Springer, 2013. doi:10.1007/978-3-642-39799-8_1.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LISK17] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017. doi:10.29007/8mwc.

[OB03] Jens Otten and Wolfgang Bibel. leanCoP: Lean connection-based theorem proving. *J. Symb. Computation*, 36(1-2):139–161, 2003. doi:10.1016/S0747-7171(03)00037-3.

[PU18] 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. doi:10.1007/978-3-319-94205-6_37.

[RBF18] 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. doi:10.1007/978-3-319-89963-3_7.

[RTdM14] 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. doi:10.1109/FMCAD.2014.6987613.

[SS18] Taro Sekiyama and Kohei Suenaga. Automated proof synthesis for the minimal propositional logic with deep neural networks. In Sukyoung Ryu, editor, *APLAS 2018*, volume 11275 of *LNCS*, pages 309–328. Springer, 2018. doi:10.1007/978-3-030-02768-1_17.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[UVv11] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP: Machine learning connection prover. In Kai Brünnler and George Metcalfe, editors, *TABLEAUX 2011*, volume 6793 of *LNCS*, pages 263–277. Springer, 2011.

[Vap00] Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Statistics for Engineering and Information Science. Springer, 2000. doi:10.1007/978-1-4757-3264-1.

[Wha16] Daniel Whalen. Holophrasm: A neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016. arXiv:1608.02644.