Explaining SAT Benchmarks

Markus Iser
Karlsruhe Institute of Technology (KIT), Institute of Theoretical Informatics, Algorithm Engineering, Germany

Abstract

Taxonomies for benchmark instances can be derived in several ways. For example, this can be based on their theoretical properties, or it can be based on their origin, e.g., a concrete application. Some algorithm selectors even generate instance classes based on a set of features in an unsupervised manner. We see a gap in the explainability of such approaches. In this paper, we present a SAT encoding for decision tree and random forest classifiers that we can use to generate prime implicants. We then apply this encoding to generate minimal explanations for two types of classifiers. The first type of classifier was trained to map SAT instances to their instance family. The second type of classifier was trained as an instance-specific SAT solver selector on a small portfolio. We show that the explainability of machine learning methods is useful for interpreting experiments as it provides valuable feedback to the algorithm developer.

2012 ACM Subject Classification Theory of computation → Logic and verification; Computing methodologies → Supervised learning by classification

Keywords and phrases Benchmarks, Explainable Algorithm Selection, Prime Implicants

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Models created by inductive learning algorithms are useful in practice. The explainability of such automatically generated models is the subject of current research projects [2, 9]. In particular, for many application scenarios where accountability concerns arise, it is imperative that what is learned can be explained [18].

Automated methods for generating explanations for machine-learned models include formalizing and encoding them in formal logic. The symbolic representation of what is learned enables the application of deductive methods for reasoning about what is learned. Deductive reasoning about inductively generated models is usually referred to by the term eXplainable AI (XAI) [1].

The term *explanation* is ambiguous. There is a plethora of possible formalizations leading to different definitions of explainability. Audemard et al. analyze the complexity of different types of XAI queries [3]. One of the explanatory queries they analyze is prime implicants.

Choi et al. present a coding of decision trees and random forests for computing prime implicants [8]. They report limitations in encoding cardinality constraints for multivalued variables.

Inductive learning methods have been used in various ways to solve SAT problems more efficiently. Cherif et al. successfully used reinforcement learning to control the switching of branching heuristics in their solver Kissat MAB [7]. Clause forgetting heuristics based on classification and regression using solver runtime data were presented by Soos et al. with their system Crystal Ball [19].

Prediction models have also been used to create algorithm selectors for portfolios of SAT solvers (cf. SATzilla [20]). In their approach known as instance-specific algorithm configuration (ISAC), Kadioglu et al. use unsuperived learning to create clusters in the feature space of SAT instances [16]. For each cluster, a configurator optimizes a SAT solver for the instances in that cluster.

Prediction models for algorithm selection induce, in a sense, a taxonomy for SAT instances. This differs from the classes used in common practice, where instances are usually assigned to an instance family [11]. The instance family may for example describe a category of SAT applications (such as *hardware verification*), or refer to a specific class of instances that is of theoretical interest (such as *pigeon hole*).

Elffers et al. analyze SAT solver configurations for instance classes of theoretical interest [10]. Audemard and Simon devise a taxonomy which is purely based on solver runtime parameters and deduce class specific solver configurations [4].

Given the circumstances sketched so far, we conclude that explanations of prediction models for SAT benchmarks can provide valuable feedback to the algorithm engineer about what was learned. Such explanations allow the algorithm engineer to reason about the induced instance taxonomy, and to improve their algorithm or configuration choices.

In this paper, we present a simple encoding of decision trees classifiers and random forests as a monotone combinatorial circuit. We show how prime implicants can be computed for the resulting formula using an off-the-shelf incremental SAT solver. Moreover, we present a tool which can encode decision tree classifiers generated with the Python package `scikit-learn` [17] into propositional formulas. Our tool uses the SAT solver `CaDiCaL` [6] via its `IPASIR` [5] interface to generate prime implicants. The prime implicants are then decoded with respect to the represented case-distinctions in the features space. We evaluate the tool on a set of 26794 SAT instances, with a large set of features including their instance families.

The document is structured as follows. In Section 2, we present a formalization of decision tree classifiers and random forest classifiers as they are implemented in the Python package `scikit-learn`. The section concludes with the required fundamentals on propositional logic, incremental SAT solvers and prime implicants. In Section 3, we present a propositional encoding of the previously formalized classifiers and an algorithm to compute prime implicants for them. We present our implementation in Section 4, where we also evaluate our approach on a classifier which was trained to associate SAT instances families for a set of 26794 SAT instances based on a set of instance features which we also describe in that section. We also evaluate our approach on a classifier which was trained to select SAT solvers from small portfolios of solvers drawn from SAT competition 2020. We conclude with Section 5.

2 Preliminaries

In the following sections, we formally describe the data structures created by *decision tree* (Section 2.1.1) and *random forest* (Section 2.1.2) classifiers which realize the learned prediction function as they are implemented in the Python package `scikit-learn` [17]. In Section 2.2, we introduce some basic notions of propositional logic in particular regarding prime implicants. The formalizations will later serve as a fundament to the propositional encoding of the prediction models under consideration (Section 3).

2.1 Classification

The *classification problem* under consideration is specified as follows. Given a set of training samples $T \subset \mathbb{R}^n$, a set of classes K, and a functional ground-truth relation $G \subset T \times K$, devise a prediction function $c : \mathbb{R}^n \to K$ which maximizes the cardinality of correctly classified training samples $\left|\{t \in T \mid (t, c(t)) \in G\}\right|$. The challenge is to devise a prediction function which generalizes to yet unseen samples of the feature space. In theory, decision trees can grow until they classify each training sample correctly, i.e., the classification result equals ground truth. In practice however, pruning techniques such as maximum depth or minimum

leaf size are used in order to improve the generalization of a classifier to yet unseen samples. An introduction to methods and challenges of classification tasks and supervised learning in general can be found in Hastie et al. [12].

2.1.1 Decision Tree Classifiers

Let a classification instance (T, K, G) and its solution in form of a *decision tree* $\mathcal{D}$ be given. A decision tree $\mathcal{D} = (V, E, f, t)$ is specified by a *binary tree* with nodes V and edges $E \subset V \times V$. The nodes are partitioned into *inner nodes* $V^I := \{v \in V \mid \exists x, (v, x) \in E\}$, and *leaf nodes* $V^L = V \setminus V^I$. The *root node* r is the special inner node with $\nexists x, (x, r) \in E$. The set of edges E is partitioned into *positive edges* E^+ and *negative edges* E^- such that each inner node $v \in V^I$ has exactly one *positive successor* $\mathsf{hi}(v) \in V$ with $\big(v, \mathsf{hi}(v)\big) \in E^+$ and exactly one *negative successor* $\mathsf{lo}(v)$ with $\big(v, \mathsf{lo}(v)\big) \in E^-$. Associated with each inner node $v \in V^I$ is a feature index $f : V^I \to \{1, 2, \ldots, n\}$ and a threshold $t : V^I \to \mathbb{R}$.

Given a set of samples $S \subseteq \mathbb{R}^n$, the decision tree $\mathcal{D}$ induces a partitioning $P_S : V \to 2^S$ of samples over nodes in V. Starting with $P_S(r) = S$ for root node r, the partitioning is recursively defined as in Equation 1.

$$P_S(v) = \begin{cases} S & \text{iff v is the root node} \\ \big\{(s_0, \ldots, s_n)^\intercal \in P_S(x) \mid s_{f(x)} \leq t(x)\big\} & \text{iff } v = \mathsf{hi}(x) \\ \big\{(s_0, \ldots, s_n)^\intercal \in P_S(x) \mid s_{f(x)} > t(x)\big\} & \text{iff } v = \mathsf{lo}(x) \end{cases} \tag{1}$$

The partitioning P_T of the set of training samples T together with ground truth G induces for each node v a probabilty distribution p_v over classes $k \in K$ as in Equation 2.

$$p_v(k) = \frac{\big|\{s \in P_T(v) \mid (s, k) \in G\}\big|}{\big|P_T(v)\big|} \tag{2}$$

The decision tree $\mathcal{D}$ specifies a classification function $c(s) : \mathbb{R}^n \to K$. Given a sample $s \in \mathbb{R}^n$, there exists exactly one leaf node $v \in V^L$ such that $s \in P_{\{s\}}(v)$. The classification result for sample s is the one with the highest probability in that node, i.e., $c(s) = \arg\max_{k \in K} p_v(k)$.

2.1.2 Random Forest Classifiers

In the context of machine learning, *bagging* is the idea to combine several weak learners to one stronger learner by using some kind of voting to determine one collective prediction [12]. A *random forest* $\mathcal{R}^d = \{(T_i, \mathcal{D}_i) \mid 1 \leq i \leq d\}$ combines a set of d decision trees. Each decision tree $\mathcal{D}_i$ is independently trained on randomly selected subsets of the training samples $T_i \subset T$.

Let a classification problem (T, K, G) and a random forest $\mathcal{R}^d = \{(T_i, \mathcal{D}_i) \mid 1 \leq i \leq d\}$ be given. For each decision tree $\mathcal{D}_i = (V_i, E_i, f_i, t_i)$, the training samples T_i induce a probability distribution $p_v(k)$ for nodes $v \in V_i$ (cf. Section 2.1.1). Given a sample $s \in \mathbb{R}^n$, for each decision tree $\mathcal{D}_i$ there exists exaclty one leaf node $v_i \in V_i^L$ such that $s \in P_{\{s\}}(v_i)$. The collective class probabilities $p'(k)$ for a sample s are determined by averaging over the class probablities in each tree as in Equation 3. The classification result for sample s is then given by $c'(s) = \arg\max_{k \in K} p'(k)$.

$$p'(k) = \frac{1}{d} \sum_{i=1}^{d} p_{v_i}(k) \tag{3}$$

2.2 Prime Implicants

In this document, propositional formulas are given in conjunctive normal form (CNF). A propositional *formula* F is defined over a finite set of Boolean variables V. Each formula is a conjunction of clauses, a clause is a disjunction of literals and a literal is either a variable or its negation. A set of (non-contradictory) literals over V is a *model* M of a formula F, iff its intersection with any clause in F is non-empty. A model is *complete* iff $|M| = |V|$ and *partial* otherwise. Each model M of a formula F is also an implicant of F, denoted by $M \models F$. An implicant M is a prime implicant of F iff $\nexists M' \subset M, M' \models F$.

Prime implicants for CNF formulas can efficiently be calculated through eager minimization of a model with an incremental SAT solver [15, 13]. If the CNF formula encodes a *monotonic* combinational circuit, it is suffient to minimize the model for the input variables of that circuit, as a complete assignment can later be deduced from the minimized input assignment [13].

3 Approach

In the following we describe CNF encodings for decision tree classifiers (Section 3.1) and random forests (Section 3.2). The encodings resemble monotonic circuits which constrain feature values at their inputs. In Section 3.3, we outline an algorithm to efficiently compute prime implicants for these formulas. In Section **??**, we show how to decode these prime implicants in order to map them to a set of case distinctions in the feature space.

3.1 Encoding Decision Tree Classifiers

Given a classification instance (T, K, G) and a corresponding decision tree $\mathcal{D} = (V, E, f, t)$, we create its propositional encoding as follows.

3.1.1 Variables

We introduce three sets of Boolean variables (α, β, and γ), which together form the set of variables V over which our encoding F is defined. For each class $k \in K$, we introduce a *class variable* $\alpha(k)$ for indicating the classification result. For each inner node $v \in V^I$, we introduce two *node variables* $\beta^+(v)$ and $\beta^-(v)$, with $\beta^+(v)$ denoting its successor $\mathsf{hi}(v)$, and $\beta^-(v)$ denoting its successor $\mathsf{lo}(v)$. For each feature f, we construct the auxilary set of thresholds σ_f as specified by Equation 4. From σ_f we construct the auxilary set of threshold intervals τ_f as shown in Equation 5.

$$\sigma_f := \{t(v) \mid f(v) = f, v \in V\} \cup \{-\infty, \infty\} \tag{4}$$

$$\tau_f := \{(t_0, t_1] \mid t_0, t_1 \in \sigma_f \wedge t_0 < t_1 \wedge \nexists t' \in \sigma_f, t_0 < t' < t_1\} \tag{5}$$

For each feature f and each threshold interval $z \in \tau_f$, we introduce the Boolean variable $\gamma(f, z)$. Variables of type γ indicate whether the respective interval is excluded by the decision tree.

3.1.2 Clauses

For each class $k \in K$, we determine the set of leaf nodes $V_k^L \subseteq V^L$ in which the classifier outputs k. Then we encode the *class constraint* for each class k as depicted in Encoding 3.1.1.

▶ **Encoding 3.1.1** (Class Constraints).

$$\forall k \in K, \alpha(k) \rightarrow \bigvee_{v \in V_k^L} \beta^+(v)$$

For each inner node $v \in V^I$, we encode the *node constraints* as shown in Encoding 3.1.2.

▶ **Encoding 3.1.2** (Node Constraints).

$$\forall v \in V^I, \beta^+(\mathsf{hi}(v)) \rightarrow \beta^+(v)$$
$$\forall v \in V^I, \beta^-(\mathsf{hi}(v)) \rightarrow \beta^+(v)$$
$$\forall v \in V^I, \beta^+(\mathsf{lo}(v)) \rightarrow \beta^-(v)$$
$$\forall v \in V^I, \beta^-(\mathsf{lo}(v)) \rightarrow \beta^-(v)$$

For each inner node $v \in V^I$, we encode its *interval constraints* as follows. We split the set of threshold variables $\tau_{f(v)}$ and into two auxiliary sets $\tau^+(v)$ and $\tau^-(v)$ which are defined as follows.

$$\tau^+(v) := \{(t_0, t_1] \in \tau_{f(v)} \mid t_1 \leq t(v)\}$$
$$\tau^-(v) := \{(t_0, t_1] \in \tau_{f(v)} \mid t_0 > t(v)\}$$

Then we encode the *interval constraints* as in Encoding 3.1.3.

▶ **Encoding 3.1.3** (Interval Constraints).

$$\forall v \in V^I, \bigwedge_{\tau \in \tau^-(v)} \beta^+(v) \rightarrow \gamma(v, \tau)$$
$$\forall v \in V^I, \bigwedge_{\tau \in \tau^+(v)} \beta^-(v) \rightarrow \gamma(v, \tau)$$

3.2 Encoding Random Forest Classifiers

For encoding random forest classifiers, we encode several decision trees. The encoding of each decision tree is similar to the encoding described in Section 3.1. In particular, the node constraints given by Encoding 3.1.2 is just the same. But as classes are not simply determined by singlular leaf nodes, we need a different encoding for class constraints.

3.2.1 Encoding Interval Constraints for Random Forests

As random forests constist of multiple decision trees, there is a much larger total number nodes. This induces a much larger number of thresholds for each feature. In initial experiments the total number thresholds for a single feature could go into the thousands. Since the interval constraints as presented in Encoding 3.1.3 introduce a clause per threshold per node, the encoding is not efficient.

We therfore use a new encoding for interval contraints which has a higher *constant* size overhead in terms of variables and clauses, but which then needs only a single clause per node and threshold and therefore scales better.

We introduce two types of variables and an implication chain to connect them. Then both types of variables are connected to one interval variables. ...

3.2.2 Encoding Class Constraints for Random Forests

Recapitulate how classification works in the random forests described in Section 2.1.2. Given a sample s, for each decision tree $\mathcal{D}_i$ with leaf nodes V_i^L, there is exactly one leaf node $v_i \in V_i^L$ such that $s \in P_{\{s\}}(v_i)$. Given the tuple of leaf nodes $(v_1, \ldots, v_d)$, i.e., one in each tree, the class of s is determined by the maximum average class probability over those leaf nodes as depicted in Equation 3.

One way to encode this, would be to encode an arithmetic circuit.[1] For the implementation, which we present in this paper, we took a simpler approach. Consider again the tuples of leaf nodes $L := V_1^L \times \cdots \times V_d^L$. To each tuple in $t \in L$, we can assign a class in K by determining the maximum average class probability for the leaf nodes. But the size of L grows exponentially with the number of decision trees in the forest. Fortunately, only a small fraction of tuples in L is actually a valid combination of leaf nodes, i.e., they also satisfy the node and interval constraints. In order to determin the set of valid tuples $L' \subseteq L$, we encode a small auxiliary SAT problem A. We enumerate the solutions of A to determine L'. For each tuple in $t_i \in L'$, we introduce a new variable δ_i and encode the *tuple constraint* as depicted in Encoding 3.2.1.

▶ **Encoding 3.2.1** (Tuple Constraints).

$$\bigwedge_{v \in t_i} \delta_i \rightarrow \beta^+(v)$$

For each tuple t_i, we determine its class $k_i \in K$. Then for each tuple t_i we encode the class constraints as depicted in Encoding 3.2.2.

▶ **Encoding 3.2.2** (Class Constraints).

$$\bigwedge_{t_i \in L'} \alpha(k_i) \rightarrow \delta_i$$

3.3 Computing Prime Implicants

Given a classifier, we encode it as described above. Then we add an explanation clause with class variables for the classes which we want shortest explanations for. In most cases, the explanation clause might be a unit clause, since we often want shortest explanations for a single class.

The encoding of a decision tree resembles a monotonic combinatorial circuit. It is rooted in the explanation clause of the classes to explain. The inputs to this circuit are represented by the interval variables. Due to monotonicity, we can compute prime implicants by minimizing assignments to the interval variables.

Algorithm 1 outlines the procedure. The algorithm receives as input the formula encoding the classifier including the explanation clause, and the set of input variables which is comprised of all interval variables.

[1] In this circuit, for each decision tree in the forest, we would encode a list of bit-vectors representing the probabilities of each class, which are encoded to be equivalent to the known class probabilities – dependent on the activated leaf node. For each class, we would further encode an adder circuit to calculate the sum of each classes probabilities over all trees in the forest. On top of that, we would then encode a circuit which ensures that the sum of probabilities of the class to explain is the maximum of all classes.

Algorithm 1 Incremental Computation of Prime Implicants

Input: CNF Formula: F
Input: Input Variables: I
Output: Prime Implicants: P

```
1  (sat, model) ← solve(F, ∅)
2  while sat do
3      while sat do
4          assumptions ← ∅
5          minim ← ∅
6          for v ∈ I do
7              if v ∈ model then
8                  minim ← minim ∪ {−v}
9              else
10                 assumptions ← assumptions ∪ {−v}
11         F ← F ∪ minim
12         (sat, model) ← solve(F, assumptions)
13         if not sat then
14             P ← P ∪ {−v | v ∈ minim}
15     (sat, model) ← solve(F, ∅)
16 return P
```

4 Evaluation

4.1 Implementation and Dataset

An implementation of our approach can be found on GitHub.[2] It consists of a `Python` module written in `C++` which computes prime implicants (cf. Section 3.3) using the incremental SAT solver `CaDiCaL` by Armin Biere [6]. The encoding in implemented in `Python` for the decision tree classifier implementation in the `scikit-learn` package [17].

4.1.1 Dataset

Our implementation accesses data via our `gbd-tools` package[3] which we orginially presented in [14]. The three datasets which we used in our evaluations can be obtained at `https://gbd.iti.kit.edu/` and are described in the following.

4.1.1.1 Meta Features

We collected a set of meta features in a huge database of more than 26k benchmark instances (mainly from SAT competitions since 2002). These meta fetures include the sat/unsat result (if known) and the instance family. The instance families have been manually deduced from the documents in SAT competition proceedings or SAT competition presentation slides.

[2] `https://github.com/Udopia/pi-explanations`
[3] `https://pypi.org/project/gbd-tools/`

4.1.1.2 Base Features

Amounts This set of features includes the number of clauses, variables, the numbers of clauses of sizes 1 to 9, the number of horn clauses, inverse horn clauses, positive clauses, and negative clauses.

Distribution over Horn Clauses For this set of features, we count for each variable its number of occurences in horn clauses. The counts are represented by five features, their mean, variance, minimum, maximum and entropy. We add another five features for the count of variable occurences in inverse horn clauses.

Balance of Literal Polarities Here we report on two distributions in terms of their mean, variance, minimum, maximum and entropy. The first distribution captures for each variable the fraction of its number of positive occurences in clauses divided by the number of negative occurences in clauses. The second distribution captures for each clause the number of positive literals divded by the number of negative literals.

Distribution of Node Degrees Degree distributions by mean, variance, minimum, maximum and entropy for nodes in the variable interaction graph, nodes in the clause graph, variable nodes in the variable-clause graph, clause nodes in the variable-clause graph.

4.1.1.3 Gate Features

Amounts total number of gates, number of root variables, number of input variables, number of generically recognized gates, number of monotonically nested gates, number of non-monotonically nested and-gates, number of non-monotonically nested or-gates, number of non-monotonically nested trivial equivalence gates, number of non-monotonically nested equiv- or xor-gates, number of non-monotonically nested full gate (=maxterm encoding) with more than two inputs

Distribution over Levels For each gate type we also determine their levels in the decoded hierarchical gate structure, for each type we report on the distribution of levels by their mean, variance, minimum, maximum and entropy.

4.2 Explaining Classification Results

- less case distinctions (leaf node level vs. number of case distinctions by prime implicant)
- only very few features responsible (no need to calculate them all)
- few prime implicants cover most of the samples
- some prime implicants for single sample

4.2.1 Instance Family Classifier

Train classifier to predict instance family for an instance given 26794 instances of SAT Competitions.

- Number of instances: 26794
- Number of instance families: 134
- Classifier Accuracy (5-fold): 0.97

todo

Figure 1 Number of Case Distinctions needed for Instance Family Classification: Decision Tree vs. Decoded Prime Implicants

todo

Figure 2 Number of Case Distinctions for Small Portfolio Classification: Decision Tree vs. Decoded Prime Implicants

4.2.2 Small Portfolios Classifier

Froleyks et al. report on best small portfolios drawn from solvers of SAT Competition 2020 [11]. Train classifier to predict fastest solver for an instance given 400 instances of SAT Competition 2020.

- Number of instances: 400
- Best 2-Portfolio: kissat unsat, relaxed newtech
- Classifier Accuracy (5-fold): 0.67

5 Conclusion

References

1 Alejandro Barredo Arrieta, Natalia Díaz Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. *CoRR*, abs/1910.10045, 2019.

2 Gilles Audemard, Steve Bellart, Louenas Bounia, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. On the computational intelligibility of boolean classifiers. In Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 74–86, 2021.

3 Gilles Audemard, Frédéric Koriche, and Pierre Marquis. On tractable XAI queries based on compiled representations. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 838–849, 2020. URL: `https://doi.org/10.24963/kr.2020/86`.

4 Gilles Audemard and Laurent Simon. Extreme cases in SAT problems. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2016.

5 Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016. URL: `https://doi.org/10.1016/j.artint.2016.08.007`.

6 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

7 Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Combining VSIDS and CHB using restarts in SAT. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual*

Conference), October 25-29, 2021, volume 210 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.CP.2021.20.

8 Arthur Choi, Andy Shih, Anchal Goyanka, and Adnan Darwiche. On symbolically encoding the behavior of random forests. *CoRR*, abs/2007.01493, 2020. URL: https://arxiv.org/abs/2007.01493.

9 Adnan Darwiche and Auguste Hirth. On the reasons behind decisions. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 712–720. IOS Press, 2020.

10 Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström, and Laurent Simon. Seeking practical CDCL insights from theoretical SAT benchmarks. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1300–1308. ijcai.org, 2018. URL: https://doi.org/10.24963/ijcai.2018/181.

11 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artif. Intell.*, 301:103572, 2021. URL: https://doi.org/10.1016/j.artint.2021.103572.

12 Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. URL: https://doi.org/10.1007/978-0-387-84858-7.

13 Markus Iser. *Recognition and Exploitation of Gate Structure in SAT Solving*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2020. URL: https://nbn-resolving.org/urn:nbn:de:101:1-202004290459566073264 8.

14 Markus Iser and Carsten Sinz. A problem meta-data library for research in SAT. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018*, volume 59 of *EPiC Series in Computing*, pages 144–152. EasyChair, 2018. URL: https://doi.org/10.29007/gdbb.

15 Markus Iser, Carsten Sinz, and Mana Taghdiri. Minimizing models for tseitin-encoded SAT instances. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 224–232. Springer, 2013. URL: https://doi.org/10.1007/978-3-642-39071-5_17.

16 Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - instance-specific algorithm configuration. In Helder Coelho, Rudi Studer, and Michael J. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 751–756. IOS Press, 2010.

17 Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

18 Chris Percy, Simo Dragicevic, Sanjoy Sarkar, and Artur S. d'Avila Garcez. Accountability in AI: from principles to industry-specific accreditation. *CoRR*, abs/2110.09232, 2021. URL: https://arxiv.org/abs/2110.09232.

19 Mate Soos, Raghav Kulkarni, and Kuldeep S. Meel. Crystalball: Gazing in the black box of SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal,*

July 9-12, 2019, Proceedings, volume 11628 of *Lecture Notes in Computer Science*, pages 371–387, 2019. URL: https://doi.org/10.1007/978-3-030-24258-9_26.

20 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008. URL: https://doi.org/10.1613/jair.2490.