Guessing Good: Applying Machine Learning to Boolean Satisfiability

Nathaniel Yazdani **Christopher A. Mackie**
Paul G. Allen School of Computer Science and Engineering
University of Washington
Seattle, WA 98105
{nyazdani, mackic}@cs.washington.edu

Introduction

Boolean satisfiability (SAT) is a classical NP-complete problem with great practical relevance. Software verifiers, bounded model checkers, and many other tools for automated reasoning work by reduction to SAT. Theoretically, SAT instances of the scale produced by such tools could require more time than the age of the universe to solve, yet modern SAT solvers often take mere seconds to days. Effective heuristics, tuned to exploit the significant structure of real-world (*i.e.*, non-random) SAT instances, make such feats possible. Despite that, ever more complex downstream applications demand ever greater scalability of SAT solvers, which struggle to keep up.

For those unfamiliar, SAT is the problem of finding truth value assignments to variables in a conjunctive normal form (CNF) sentence of propositional logic (*i.e.*, a conjunction of disjunctions of variables or their negations)[1]. Figure 1a shows the grammar of a CNF formula, and fig. 1b presents several examples of CNF formulae.

$$
\begin{aligned}
formula &::= clause \wedge formula \\
&\mid clause \\
clause &::= literal \vee clause \\
&\mid literal \\
literal &::= variable \mid \neg variable
\end{aligned}
$$

(a) Grammar of CNF formulae.

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \quad (1)$$

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3) \quad (2)$$

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \quad (3)$$

(b) Examples of CNF formulae.

Figure 1: Illustration of conjunctive normal form (CNF).

Modern SAT solvers are based around a smart yet still brute-force algorithm called conflict-driven clause learning (CDCL), shown in alg. 1. CDCL works by randomly guessing variable assignments at branching points until it either solves the formula or reaches a conflict (incompatible assumptions). At this time, CDCL performs conflict analysis (not discussed here) to derive a *conflict clause*, which optimally summarizes the reason for the conflict in terms of which assumptions (previous branching choices) were incompatible. Conflict clauses are key to efficiently solving SAT, because they allow the solver to learn more and more nonobvious facts about the current formula as its search progresses.

The two kinds of heuristics commonly used by SAT solvers to enhance CDCL are those for *variable selection*, which rank variables in order to optimize the sequence of branching points, and those for *random restarts*, which notify (with some random influence) the solver when to abondon its current set of variable assignments and start over. In the process of solving a nontrivial CNF formula, a

[1]Technically, SAT is the decision problem for the existence of such a satisfying assignment, but modern SAT solvers accomplish this by search for a constructive proof in the form of such a satisfying assignment.

Algorithm 1 Conflict-driven clause learning (CDCL).

procedure CDCL($formula$)
  $model \leftarrow \{\}$ ▷ Variable assignments
  $level \leftarrow 0$ ▷ Decision level
  $model \leftarrow$ PROPAGATE($formula, model$) ▷ Assume unit clauses are true
  if $\exists x_i.\{x_i \mapsto \top, x_i \mapsto \bot\} \subseteq model$ **then**
    return UNSAT ▷ Formula is self-contradictory
  end if
  while INDETERMINATE($formula, model$) **do** ▷ Neither SAT nor UNSAT yet
    $level \leftarrow level + 1$
    $variable \leftarrow$ PICKVARIABLE($formula, model$) ▷ Choose next variable to branch
    $model \leftarrow model \cup \{variable \mapsto$ FLIP$(\top, \bot)\}$ ▷ Random truth assignment
    $model \leftarrow$ PROPAGATE($formula, model$) ▷ Assume any implied assignments
    while $\exists x_i.\{x_i \mapsto \top, x_i \mapsto \bot\} \subseteq model$ **do**
      $(level', clause) \leftarrow$ ANALYZECONFLICT($formula, model$)
      $formula \leftarrow formula \wedge clause$
      if $level' < 0$ **then**
        return UNSAT ▷ Can no longer backtrack
      else
        $model \leftarrow$ BACKTRACK($formula, model, level'$)
        $level \leftarrow level'$
      end if
    end while
  end while
  return SAT($model$)
end procedure

SAT solver may produce Gigabytes worth of conflict clauses, so SAT solvers also typically employ heuristics to decide which conflict clauses to remember and which to forget (and when). Unfortunately, there are no good heuristics for choosing a truth assignment at branching points, leaving SAT solvers to make uninformed, random choices. We see this as a lost opportunity.

Overview

In this project, we explore an application of machine learning as a predictor of likely variable assignments given information available to a SAT solver. A SAT solver could leverage such a predictor as a heuristic to replace the random truth assignment FLIP$(\top, \bot)$. We explored two models for such a predictor, explained shortly, that accept either features of the input formula or the entirety of the input formula. Both predictors output (modulo possible post-processing) a vector $\widehat{y} \in [0, 1]^n$, where n is the number of variables in the input formula. Each $\widehat{y}_i$ is a prediction of the (pseudo-)probability that variable x_i should be assigned true (and similarly for $1 - \widehat{y}_i$ and assigning x_i false).

For our data set, we used the application (*i.e.*, real-world, non-random) SAT instances from the SATLIB benchmark suite[2]. We chose this benchmark suite because the formulae are representative of a variety of real-world applications of SAT yet not so large as to prohibit effective processing and analysis. We instrumented MiniSAT[3] (a modern, high-performance SAT solver) to enumerate minimized[4] satisfying assignments and log all conflict clauses.

To train and test the predictor models, we needed a particular satisfying assignment for each formula in the data set to serve as its label. A formula's satisfying assignments may differ significantly, but the set of satisfying assignments $\{\alpha_1, \ldots, \alpha_n\}$ may typically be partitioned into disjoint subsets

[2] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

[3] `http://minisat.se/`

[4] "Minimization" in this sense means that the solver tries to reduce noise in its assignments due to inconsequential variables' assignments.

$S_1, \ldots, S_k$ such that the variable assignments $\widehat{\alpha}_i = \bigcap_{\alpha_j \in S_i} \alpha_j$ are sufficient to satisfy the formula, despite assigning no truth value to some variables.

Ideally, we could use such an $\widehat{\alpha}_i$ as the label for each formula, in order to grant the prediction model maximal flexibility during training by ignoring predictions for variables "irrelevant" to a particular formula (according to the label). We investigated this but found it intractable with our computational resources: enumerating all satisfying assignments to our benchmarks consumed hundreds of Gigabytes, which we had little hope of processing effectively. We explored using a binary decision diagram (a data structure to compactly represent a decision tree) to partition the set of satisfying assignments as they are enumerated, but this turned out to be computationally intractable due to the typical number of variables in a formula (≈ 1000).

In the end, we chose to use the first satisfying assignment enumerated by MiniSAT, because the first satisfying assignment is likely — though not necessarily — simpler than those enumerated later.

Approach

Variable-based Model

Our first model bases its prediction for a variable on features of that particular variable in the context of the input formula. We made this predictor operate on each variable separately in order to maximize its flexibility, *e.g.*, avoid fixing the number of supported variables and/or clauses ahead of time. As an added benefit, SAT solvers already track some statistics about each variable in the context of the input formula for the purpose of variable selection. (So, we also knew that this kind of information is useful.)

We transform each variable into a feature vector, and consider each feature vector as a different data point. In this way, our variable-based model does not work at the granularity of the whole input formula, but of variables. This gains better performance and flexibility but loses any sense of correlation between multiple variables. To combat this somewhat, we demean all feature vectors originating from the same inoput formula.

To produce the feature vector, we first iterate over $L = \{-x, x\}$, the literals of x. For each $l \in L$, we then iterate over $C = \{\phi_1(l), \phi_2(l), \phi_3(l), \phi_4(l), \phi_{\text{horn}}(l), \phi_{\text{cohorn}}(l), \phi(l)\}$, subsets of clauses which contain l. To make our notation clear, $c(l)$ is a subset of clauses from the set c, which contain l. The sets ϕ_i are subsets of clauses from the formula with exactly i literals. The set ϕ_{horn} is the set of clauses which have at most one positive literal, while ϕ_{cohorn} is the set of clauses which have at most one negative literal. ϕ is simply the set of all clauses in the formula. For each $c \in C$, we then calculate the following features for the literal-clause pair (l, c):

1. $|c|$, the number of clauses in c
2. $\sum_i \frac{1}{|c_i|}$, where $|c_i|$ is the number of literals in clause c_i
3. $\max_i |c_i|$, the maximum size of any clause in c
4. $\min_i |c_i|$, the minimum size of any clause in c
5. $\sum_i 2^{-|c_i|}$
6. The number of clauses in c which contain more positive literals than negative
7. The number of clauses in c which contain more negative literals than positive
8. The number of clauses in c which contain an equal number of positive and negative clauses

For a formula F, we extract a feature matrix $X \in \mathbb{R}^{v \times f}$ where c is the number of variables in F, and $f = 112$ is the number of features extracted for each variable. Similarly, for each formula in our training and test sets, we receive a variable assignment vector $y \in \{0, 1\}^v$, where 0 is false, and 1 is true. Our chosen model for variable based analysis is logistic regression. In order to introduce some manner of correlation between variables, we first demean X and exclude w_0 from our model. Therefore, our model simply consists of weights $w \in \mathbb{R}^f$ for the features. Our model can then predict variable assignment likelihoods by computing $\widehat{y} = \text{sigmoid}(Xw)$. We implemented this model in as a neural net with a fully connected initial layer, followed by a dropout layer and a sigmoid activation.

Clause-based Model

Our second model bases its predictions on the sequence of clauses that constitute the formula; in other words, it observes the entire input formula, though not all at once. We chose to make this model iterative like this in order to avoid fixing the number of clauses and avoid a blow-up in the number of weights (if the model were to receive an encoding of the formula all at once). Since this model receives the entire formula, its outputs its predictions for all variables — the number of which we fixed to 1000 (adequate for our data set) — simultaneously. Because the model operates on a *sequence* of clauses, this model could improve its variable-assignment predictions *online*, by consuming conflict clauses as the solver discovers them. Such an application would allow the predictor to retain some information from every conflict clause learnt by the solver, even if the solver itself must forget some due to memory constraints.

Since this predictor operates over sequences, we chose to implement it with a recurrent neural network. We encoded each clause as a vector $\mathbf{c}$ such that $\mathbf{c}_i = -1$ if the clauses contains $\neg x_i$, $\mathbf{c}_i = 1$ if the clauses contains x_i, and $\mathbf{c}_i = 0$ if the clause contains neither $\neg x_i$ nor x_i. Since leveraging conflict clauses is such a compelling advantage of this model, we chose to incorporate them into the training data. We implemented the model as a long-short-term-memory (LSTM) recurrent neural network, so that the predictor may propagate information besides its previous predictions through time. We also used drop-out to mitigate the possibility of overfitting.

We had to modify our handling of the data set slightly in order to train this model, though the actual predictor can hide the pre- and post-processing from the user. Specifically, we fixed the number of supported variables to 1000, and represented a clause as a row vector $c \in \{-1, 0, 1\}^v$ where -1 indicates a negative literal, 0 indicated a variable is not used, and 1 indicated a positive literal. Since we had to pad clauses with irrelevant variables, we also had to pad label vectors with irrelevant variables.To accomplish this, we changed the representation slightly so that $y \in -1, 0, 1^v$, where -1 indicates false, 0 indicated that the variable has no assignment, and 1 indicates true. Furthermore, for the purposes of training, we had to fix the number of clauses in a data point to 5000. To accomplish this, we appended as many conflict clauses from our training set as possible to formula clauses, and prepended rows of zeros if there was still room. This resulted in data points $X \in \{-1, 0, 1\}^{c \times v}$ where $c = 5000$ and $v = 1000$, with label vectors $y \in \{-1, 0, 1\}^v$. Note that, unlike with the variable based model, we keep formulas as single data points, rather than split them into several variables.

Evaluation

Given the logistic nature of the problem space, we chose to use a binary accuracy to judge our effectiveness. It is important to note that since our two models required slightly different input dimensions with regard to what is considered a single data point, the two accuracies are not calculated in the exact same way, and cannot be interpreted the same way.

While it is convenient to think of the variable based model as operating on an entire formula, in reality it operates on the level of variables. That is, we could feed our model with a single variable's feature vector, and receive the probabilty of that variable being true. This means that when we aggregate several formulae together during training, or testing, our model cannot distinguish between different formulae; it sees only variables. Therefore, the mean binary accuracy is calculated for each datapoint (i.e. each variable), and the mean of this value across all data points is reported. However, since each datapoint has a single, scalar label, the mean binary accuracy for each data point is simply the binary accuracy of the label, which is 0 or 1. The mean of all such variables is the proportion of variables that the model gets right in aggregate. We formalize this below such that $\hat{y} \in [0, 1]^n$ is the vector of predictions made by the model across all data points, and $y \in \{0, 1\}^n$ is the vector of all labels:

$$
\begin{aligned}
A_{\text{test}} &= \frac{1}{n} \sum_{i=1}^{n} A(y_i, \hat{y}_i) \\
&= \frac{1}{n} \sum_{i=1}^{n} |y_i - \text{round}(\hat{y}_i)|
\end{aligned}
$$

Meanwhile, due to the constraints of LSTM, we have set the dimensions of every datapoint and label vector for the clause based model. That is, we pad all clause vectors to be in $\{-1, 0, 1\}^{1000}$, and therefore we pad all label vectors to be in $\{-1, 0, 1\}^{1000}$. For a given datapoint, our model actually produces an entire label vector, rather than the single probability given by the variable based model. This means that computing the mean binary accuracy for a single label is not simply the binary accuracy of the label, it really is a mean across multiple accuracies. This means that the reported accuracy for the clause based model is a mean of means. We formalize this below such that $\hat{Y} \in [-1, 1]^{n \times 1000}$ is the matrix of predicted labels for all n data points, and $Y \in \{-1, 0, 1\}^{n \times 1000}$ is the matrix of true labels for all n data points:

$$\begin{aligned} A_{\text{test}} &= \frac{1}{n} \sum_{i=1}^{n} A(y_i, \hat{y}_i) \\ &= \frac{1}{n} \sum_{i=1}^{n} \frac{1}{1000} \sum_{j=1}^{1000} |Y_{ij} - \text{round}(\hat{Y}_{ij})| \end{aligned}$$

While the disparity between the two metrics makes it difficult to compare our models, they are still each fair indicators of success. As such, we feel it is appropriate to still report our results. To evaluate our models, we selected several subsets of our SATLIB data as different datasets. Within a dataset, we randomly split data into 80% training data, and 20% testing data. Below is a table of our results:[5]

Dataset	Variable Mean Accuracy	Clause Mean Accuracy	Description
`*`	68.3	75.1	Full SATLIB dataset.
`mini`	66.7	84.2	Small subset for preliminary results.
`flat`	66.7	76.8	Flat graph coloring instances.
`sw`	80.0	50.0	Morphed graph coloring instances.
`ii`	82.4	75.2	Inductive inference instances.

Figure 2: A table of model performance for different subsets of SATLIB.

Conclusion

We believe that we have demonstrated that machine-learning can make predictions useful to a modern SAT solver. One may be surprised by the accuracies of our predictors, given that SAT is an NP-complete problem, but such results are actually quite reasonable. Application-generated SAT instances are known to have significant structure to them, and every effective heuristic in SAT solving exploits this in some way. Sometimes the structure is obvious; for instance, Horn clauses (clauses including at most one non-negated variable) are prevalent and efficient to solve. Heuristics for variable selection exploit properties of a SAT formula's underlying, hidden community structure, though, humorously, VSIDS was devised without even knowing this. Machine learning is successful for this problem because machine learning excels at finding (and exploiting) such hidden structure in input data, even when a human might have difficulty identifying and understanding the structure and its significance. For instance, our results clearly demonstrate that our models perform much better when the dataset is constrained to a particular application. This allows our models to pinpoint the inherent structure in these instances, that human's cannot quite identify. Meanwhile, while our models still do reasonably well with more general datasets, they are not nearly as effective. We have demonstrated that machine learning can play a role in providing additional resources to SAT solvers, in order to improve their random guesses.

References

[1] Audemard, G., Fischmeister, S., Ganesh, V., Newsham, Z., & Simon, L. (2014). Impact of Community Structure on SAT Solver Performance. *SAT*.

[2] Blaschko, M.B., & Flint, A. (2012). Perceptron Learning of SAT. *NIPS*.

[5] We reported the `mini` results in our poster, but we misread VMA as 73.7.

[3] Czarnecki, K., Ganesh, V., Liang, J.H., & Poupart, P. (2016). Learning Rate Based Branching Heuristic for SAT Solvers. *SAT*.

[4] Ganesh, V., Near, J.P., Rinard, M., & Singh, R. (2009). AvatarSAT: An Auto-Tuning Boolean SAT Solver. *MIT Technical Report*.